D-BOOKER
éditions

2e édition

DÉVELOPPER DES JEUX AVEC GODOT 4

et le langage C#

Anthony Cardinale

Développer des jeux

avec Godot 4 et le langage C#

par
Anthony Cardinale
avec la contribution de Julian Murgia

D-BOOKER
éditions

Développer des jeux • avec Godot 4 et le langage C#
par Anthony Cardinale, avec la contribution de Julian Murgia

ISBN (PDF) : 978-2-8227-1107-4

Copyright © 2023 Éditions D-BookeR
Tous droits réservés

Conformément au Code de la propriété intellectuelle, seules les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective ainsi que les analyses et les courtes citations dans un but d'exemple et d'illustration sont autorisées. Tout autre représentation ou reproduction, qu'elle soit intégrale ou partielle, requiert expressément le consentement de l'éditeur (art L 122-4, L 122-5 2 et 3a).

Publié par les Éditions D-BookeR, 229 rue Solférino, 59000 Lille
www.d-booker.fr
contact@d-booker.fr

Les exemples (téléchargeables ou non), sauf indication contraire, sont propriété des auteurs.

Conception de la couverture : D'après une création de Marie Van Der Marlière (www.marie-graphiste.com)
Logo de Godot Engine : © Andrea Calabró (Creative Commons Attribution License [CC-BY 3.0])
Mise en page : Générée sous Calenco avec des XSLT développées par la société NeoDoc (www.neodoc.biz)

Date de publication : Sept. 2023
Édition : 2
Version : 2.0

Table des matières

À propos des auteurs viii
Avant-propos ix
1. Qu'allons-nous voir dans ce livre ? ix
2. Crédits photos xii
3. Sources des exemples xii
4. Terminologie en anglais et interface française xii
5. URL raccourcies xiii

Notions fondamentales 1

1. Découverte de Godot Engine 2
1.1. Téléchargement de Godot 4
1.2. L'interface de Godot 5
1.3. Modification de l'interface 15
1.4. Nœuds et arbres 16

2. Création d'une scène sous Godot 18
2.1. Organisation du projet 19
2.2. Import des textures 20
2.3. Création d'une balle physique 23

3. L'instanciation avec Godot 33
3.1. Création de la plateforme 33
3.2. Création de la scène principale 36
3.3. Instanciation de scènes dans Godot 37
3.4. Lancement du jeu 39

4. Initiation à la création de scripts avec Godot 41
4.1. Premier script 42
4.2. Indentation 46
4.3. Variables et constantes 46
4.4. Commentaires 47
4.5. Conditions 47
4.6. Afficher du texte dans la console 48

5. Plus loin avec C# 49
5.1. Création d'un bouton 49
5.2. Récupérer un texte saisi par l'utilisateur 53

6. Interview de Gilles Roudière, contributeur au projet Godot 55

Développement d'un jeu 2D 60

7. Mise en place du projet 61
7.1. Création du projet 61
7.2. Trouver des ressources pour son jeu 62
7.3. Importer les ressources 64
7.4. Paramètres du projet 67
7.5. Système de coordonnées 2D 68

8. Création du personnage joueur 69
8.1. Création du nœud joueur 69
8.2. Création du script du personnage 72

9. Animation du personnage 82
9.1. Création des animations 83
9.2. Programmation des animations 87

10. Création d'une plateforme 2D 90
10.1. Création d'une nouvelle scène 90
10.2. Création de la plateforme 91
10.3. Programmation de la gravité 93
10.4. Programmation du saut 96

11. Mise en place d'un TileSet 98
11.1. Création des tuiles 100
11.2. Solidité des tuiles 103

12. Conception d'un niveau du jeu 108
12.1. Création de la structure de notre niveau 108
12.2. Mise en place de la caméra 110
12.3. Création d'un arrière-plan en parallaxe 112

13. Interaction avec les objets 118
13.1. Création d'une pierre précieuse 118
13.2. Création du script de ramassage 121
13.3. Test de notre script 127

14. Création de l'interface utilisateur 129
14.1. Mise en place de l'interface 130
14.2. Programmation de l'interface 136

15. Ajout des ennemis 139
15.1. Création d'un ennemi statique 139
15.2. Création d'un ennemi mouvant 142

16. Musique et effets sonores 149
16.1. Musique d'ambiance 149
16.2. Effets sonores 152

17. Finalisation de notre jeu 156
17.1. Création du menu principal 156
17.2. Script du menu principal 160
17.3. Compilation du projet 162

Création d'un jeu 3D 167

18. Modélisation 3D : Initiation à Blender 168
18.1. Téléchargement de Blender 169
18.2. L'interface de Blender 170
18.3. Les raccourcis et outils indispensables 171
Création rapide 173
Changer de mode 174
Points, arêtes, faces 174
Les transformations 175
Mode Transparence 177
Couper le modèle 3D 178
Outils de sélection 179
Extrusion 181
Biseau 181
Récapitulatif 182

19. Modélisation 3D du niveau de notre jeu..... 183
19.1. Création des zones 183
19.2. Alignement des deux zones 189
19.3. Création des couloirs 191
19.4. Isoler des éléments 196
19.5. Exporter pour Godot 198

20. Mise en place du projet Godot 199
20.1. Importation de la modélisation 200
20.2. Coloration du niveau 201
20.3. Ajout de la physique 205

20.4. Création de la scène du premier niveau 208

21. Création de la balle 211
21.1. Création de l'objet 211
21.2. Préparation des collisions 217
21.3. Instanciation de la balle et redimensionnement 219
21.4. Ajout d'une caméra 222

22. Anticrénelage, éclairage et post-processing 225
22.1. Anticrénelage 225
22.2. Éclairage 227
22.3. Post-traitement 231

23. Déplacement de la balle 238
23.1. Script de la balle 238
23.2. Script de la caméra 245

24. Déclenchement de la fin du niveau 247
24.1. Détecter la fin du niveau 248
24.2. Gérer la chute dans le vide 252

25. Quelques objets à ramasser 254
25.1. Création d'un objet à ramasser 254
25.2. Script de collision 258

26. L'interface utilisateur 264
26.1. Affichage du compteur de cubes et du temps 264
26.2. Programmation du compteur 266
26.3. Programmation du chronomètre 268

27. Finalisation et publication du jeu 272
27.1. Émettre un son quand un objet est ramassé 272
27.2. Compilation du jeu 276

Aller plus loin avec Godot 280

28. Les pistes pour s'améliorer 281

Concepts et termes anglais 284

Index 288

À propos des auteurs

Anthony Cardinale

Anthony Cardinale est ingénieur en informatique et développeur de jeux certifié. Il conçoit depuis quinze ans toutes sortes d'applications (jeux vidéo, logiciels, sites web) dans le cadre de ses différentes missions. Curieux et enthousiaste, il aime expérimenter et partager ses connaissances sur les sujets qui le passionnent. Il est l'auteur de plusieurs livres et de très nombreuses heures de formation vidéo.

AUTRES OUVRAGES :

Création d'assets 3D pour le jeu vidéo avec Blender

Initiation à la création de jeux vidéo en Lua avec Löve2D

Créez des jeux de A à Z avec unity

Créer un jeu vidéo sans coder avec Unity

avec la contribution de Julian Murgia

Julian Murgia est Docteur en informatique, spécialisé dans le traitement d'images et la détection d'objets mobiles. Passionné de jeux vidéo, il contribue diversement au projet Godot Game Engine, en tant que mainteneur de l'outil Escoria, coauteur de la documentation et membre du comité directionnel.

Avant-propos

Ce livre aborde la création de jeux avec Godot Engine de façon pratique. Après une introduction aux concepts de base et à la programmation avec C#, nous développerons deux projets complets : un jeu 2D en vue de côté et un jeu 3D. Cette approche vous aidera à assimiler beaucoup plus rapidement les fondamentaux de la création de jeux sous Godot. Après avoir lu ce livre, vous aurez toutes les clés en main pour développer vos propres jeux.

1. Qu'allons-nous voir dans ce livre ?

Chapitre 1 - Découverte de Godot Engine

Ce chapitre vous présente Godot, son interface et des généralités sur la création de jeux.

Chapitre 2 - Création d'une scène sous Godot

Le chapitre explique comment créer une scène sous Godot. Une scène correspond en général à un niveau de jeu ou à un élément de jeu comme un personnage que l'on va configurer pour pouvoir le réutiliser.

Chapitre 3 - L'instanciation avec Godot

Ce chapitre vous initie à l'instanciation, procédé permettant d'inclure des scènes dans une autre scène.

Chapitre 4 - Initiation à la création de scripts avec Godot

Ce chapitre présente le scripting sous Godot.

Chapitre 5 - Plus loin avec C#

Ce chapitre permet d'aller un peu plus loin avec C#, le langage de programmation que nous utiliserons.

Chapitre 6 - Interview de Gilles Roudière, contributeur au projet Godot

Dans ce chapitre, Gilles Roudière partage son expérience en tant que contributeur du projet Godot.

Chapitre 7 - Mise en place du projet

Ce chapitre, qui ouvre la partie Développement d'un jeu 2D, vous permettra de mettre en place le projet d'exemple qui y sera réalisé.

Chapitre 8 - Création du personnage joueur

Ce chapitre est axé sur la création de notre personnage principal.

Chapitre 9 - Animation du personnage

Ce chapitre vous apprend à créer des animations basiques sous Godot.

Chapitre 10 - Création d'une plateforme 2D

Ce chapitre vous apprend à créer des plateformes sur lesquelles le personnage pourra sauter et se déplacer.

Chapitre 11 - Mise en place d'un TileSet

Ce chapitre présente l'outil de Tilemap/Tileset et l'utilisation de tuiles.

Chapitre 12 - Conception d'un niveau du jeu

Ce chapitre explique comment créer un niveau avec les tuiles d'un Tileset.

Chapitre 13 - Interaction avec les objets

Ce chapitre explique comment créer des objets qui peuvent être ramassés par le personnage.

Chapitre 14 - Création de l'interface utilisateur

Ce chapitre montre comment afficher le nombre d'objets collectés par le joueur.

Chapitre 15 - Ajout des ennemis

Ce chapitre explique comment créer des ennemis statiques et mouvants.

Chapitre 16 - Musique et effets sonores

Ce chapitre vous explique comment mettre du son dans vos jeux.

Chapitre 17 - Finalisation de notre jeu

Ce chapitre montre comment créer le menu principal puis compiler le jeu pour pouvoir le partager.

Chapitre 18 - Modélisation 3D : Initiation à Blender

Ce chapitre introduit la partie Création d'un jeu 3D. Il permet de se familiariser avec l'interface de Blender, logiciel de modélisation 3D libre et gratuit.

Chapitre 19 - Modélisation 3D du niveau de notre jeu

Ce chapitre vous apprend à modéliser le niveau de votre jeu 3D sous Blender.

Chapitre 20 - Mise en place du projet Godot

Ce chapitre présente la mise en place du projet de jeu 3D et la préparation du niveau 1.

Chapitre 21 - Création de la balle

Ce chapitre montre comment créer la balle qui sera contrôlée par le joueur pour explorer le niveau.

Chapitre 22 - Anticrénelage, éclairage et post-processing

Ce chapitre vous apprendra des techniques d'optimisation des graphismes et du rendu.

Chapitre 23 - Déplacement de la balle

Ce chapitre vous aidera à programmer la balle, personnage principal de notre jeu 3D, afin que le joueur puisse explorer le niveau.

Chapitre 24 - Déclenchement de la fin du niveau

Ce chapitre explique comment déclencher la fin du niveau ou le game over.

Chapitre 25 - Quelques objets à ramasser

Ce chapitre présente la création d'objets 3D à ramasser et les collisions avec ceux-ci.

Chapitre 26 - L'interface utilisateur

Ce chapitre est consacré à l'interface utilisateur du jeu 3D. Il montre comment mettre en place un compteur et un chronomètre.

Chapitre 27 - Finalisation et publication du jeu

Ce chapitre termine notre dernière partie. Nous compilerons notre jeu afin de pouvoir le partager.

2. Crédits photos

Certaines illustrations de ce livre sont issues de jeux développés par des tiers : Figure 9.1 Rayman (Ubisoft, reproduite avec leur aimable autorisation) ; Figure 6.3 (Color Gray Games, reproduite avec leur aimable autorisation) ; Figure 14.1 Yo Frankie! (Blender Foundation, licence CC) ; Figure 18.1 Neverball (licence CC).

3. Sources des exemples

Vous pouvez télécharger toutes les sources des exemples de ce livre sur le site de l'auteur.

4. Terminologie en anglais et interface française

Dans le domaine du jeu et de la modélisation 3D, de nombreux termes anglais sont couramment utilisés en français. Si certains ont des correspondances en français (exemple : layer), d'autres sont sans équivalent (exemple : material, sprite). S'ajoute à cela l'emploi de termes génériques pour désigner des types de composants. Ainsi, un Shape désignera tous les composants de type Shape. Si vous débutez dans la création de jeu, vous serez sans doute un peu dérouté au premier abord, mais en pratiquant Godot ces termes vous deviendront familiers. Pour vous aider à vous repérer, nous préciserons de temps à autre, entre crochets, l'équivalence entre termes français et anglais. Aidez-vous aussi des captures d'écran et reportez vous au glossaire en fin de livre.

Par ailleurs, pour vous simplifier la tâche, nous avons pris le parti de vous présenter Godot sur la base de son interface française. Celle-ci étant maintenue par une équipe de bénévoles, qui ont dû faire face aux nombreux changements induits par l'arrivée de Godot 4, elle peut présenter ponctuellement quelques carences et varier selon la version du logiciel utilisée. Elle n'en reste pas moins d'un abord beaucoup plus intuitif pour ceux qui découvrent à la fois le logiciel et la création de jeux.

5. URL raccourcies

Dans un souci de lisibilité, et pour pouvoir les maintenir à jour, nous avons pris le parti de remplacer toutes les adresses internet par ce qu'on appelle des URL raccourcies. Une fois que vous avez accédé à la page cible, nous vous invitons à l'enregistrer avec un marque-page si vous souhaitez y revenir fréquemment. Vous disposerez alors du lien direct. Si celui-ci se périme, n'hésitez pas à repasser par l'URL raccourcie. Si cette dernière aussi échoue, vous pouvez nous le signaler !

Notions fondamentales

Cette partie est axée sur la découverte de Godot Engine et de son mode de fonctionnement. Elle a pour but de mettre tout le monde à niveau avant de partir sur la création de jeux. Nous nous familiariserons avec l'interface et nous ferons quelques manipulations pour mettre en place une scène basique (une balle rebondissante sur une plateforme). Nous verrons également comment créer des scripts sous Godot.

1

Découverte de Godot Engine

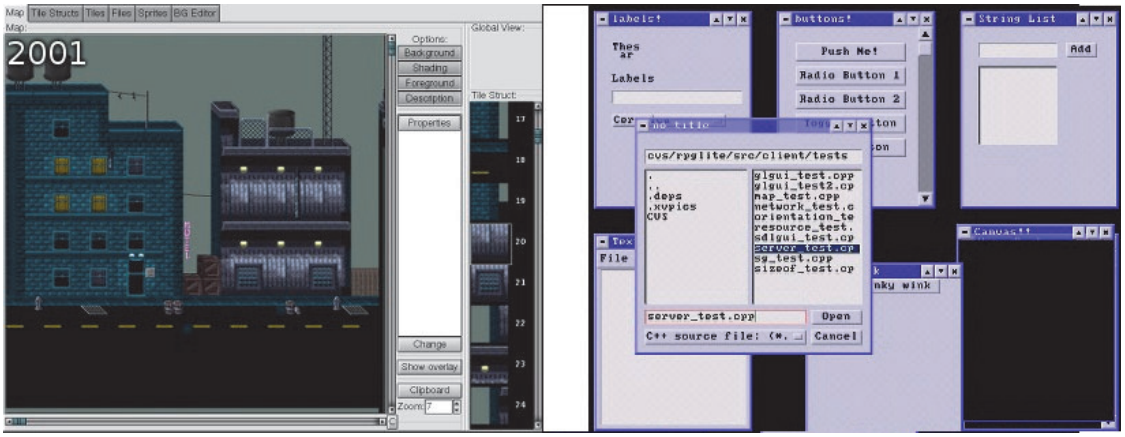
Au début des années 2000, les moteurs de jeux tels qu'on les connaît aujourd'hui n'existent pas, du moins pas sur le marché. Les seuls pseudo-moteurs existant ne sont que des outils très spécifiques qui, pour beaucoup, ne servent qu'à créer des jeux de tir à la première personne (FPS) tel que Doom.

Note › *Doom est le premier FPS à avoir eu un succès planétaire. Suite à ce succès, de nombreux développeurs ont tenté de développer des jeux similaires, ce qui a conduit à créer l'appellation* Doom-Like. *De nombreux outils de création de FPS ont alors vu le jour.*

Pour disposer d'un outil plus généraliste, Juan Linietsky et Ariel Manzur se sont lancés dès 2001 dans le développement d'un moteur de jeu qu'ils appelleront Godot en référence à la pièce de théâtre *En attendant Godot*. Ce nom a été choisi car un tel projet, s'il aboutit, demandera forcément beaucoup de temps de développement.

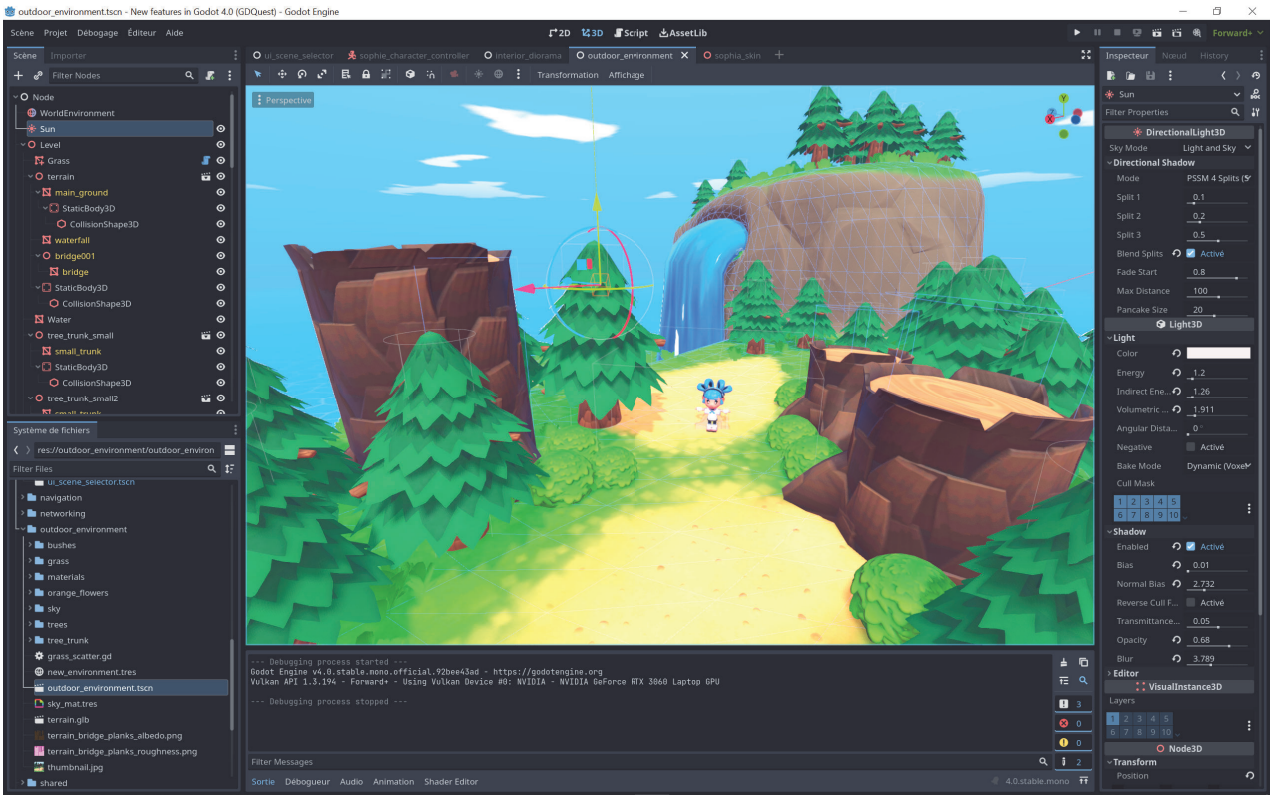
Juan Linietsky et Ariel Manzur avaient pour idée de rendre leur moteur de jeu libre dès le départ mais ils ont préféré le garder dans un premier temps propriétaire jusqu'à ce qu'il devienne mature. Après des années de travail acharné, le moteur gagne petit à petit en maturité comme on peut le voir sur les images partagées par les fondateurs sur leur blog.

Figure 1.1 : *Godot Engine en 2001*



C'est en janvier 2014 que Godot devient libre et accessible gratuitement en téléchargement via le site officiel (godotengine.org) ou via la plateforme Steam depuis 2016.

Figure 1.2 : *Godot Engine en 2023*

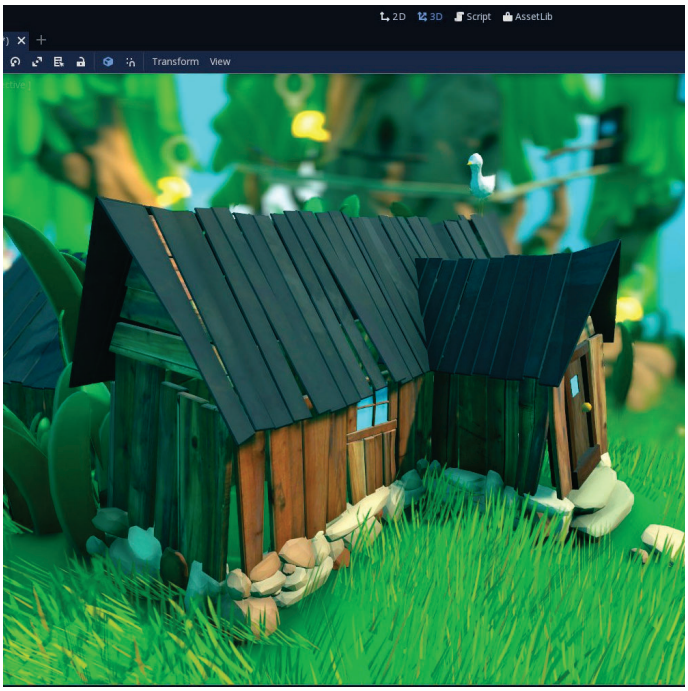


À la différence de la plupart de ses concurrents, Godot Engine a l'avantage d'être totalement gratuit et open-source. D'autres moteurs de jeux, propriétaires, vous imposent d'acheter une licence ou de reverser une partie des revenus générés par vos jeux. Godot, pour sa part, est distribué sous licence libre MIT. Cette licence vous permet de télécharger, utiliser et créer gratuitement avec Godot. Vous pouvez accéder au code source, le modifier et utiliser le logiciel à n'importe quelle finalité. Vous êtes propriétaire de vos créations et vous ne devez rien à personne. Vous pouvez distribuer ou vendre vos jeux librement.

Godot se démarque aussi par sa puissance. Il utilise les dernières techniques de rendu pour proposer des graphismes très poussés (voir Figure 1.3). Il permet de créer des jeux 2D et 3D à destination des principales plateformes (PC, mobiles et consoles).

Depuis 2014, Godot a beaucoup évolué pour devenir un environnement de développement intuitif et performant. L'interface très complète vous permet de paramétrer vos assets (modèles 3D, sons, textures...), de créer vos niveaux, de programmer vos scripts et de tester le tout à la volée. Dans sa version 4, Godot a frappé un grand coup en se rapprochant de ce qui se fait de mieux en matière de moteur de jeu et de simplicité d'utilisation même pour les débutants.

Figure 1.3 : *Exemple de rendu 3D sous Godot*



Aujourd'hui, Godot est de plus en plus utilisé et tend à devenir un incontournable dans le développement de jeux. Que vous soyez développeur de jeux, entrepreneur ou simplement passionné par le développement de jeux, Godot est un excellent choix pour concevoir des jeux de grande qualité de façon intuitive.

1.1. Téléchargement de Godot

Note *› Ce livre est basé sur la version 4.0 du logiciel. Au moment où vous le lirez, de nouvelles versions seront peut-être sorties. Cela ne devrait pas poser de problèmes, les versions 4.1.x comme par exemple 4.1.2 n'impliquent que des corrections de bogues.*

Pour commencer, nous allons télécharger Godot. La meilleure façon est de le faire directement via le site officiel. Cliquez ensuite sur le bouton DOWNLOAD et choisissez la version qui correspond à votre système d'exploitation.

Téléchargez la version .NET (compatible avec le langage C#) et non la version standard (uniquement compatible avec GDScript). Ce livre étant basé sur des exemples C#, il vous faudra obligatoirement utiliser cette version .NET. Le C# a été introduit à partir de la version 3 de Godot et est à présent pleinement utilisable. Il permet d'obtenir de meilleures performances.

Une fois le fichier ZIP téléchargé, il ne vous reste plus qu'à l'extraire pour pouvoir accéder à l'exécutable de Godot Engine.

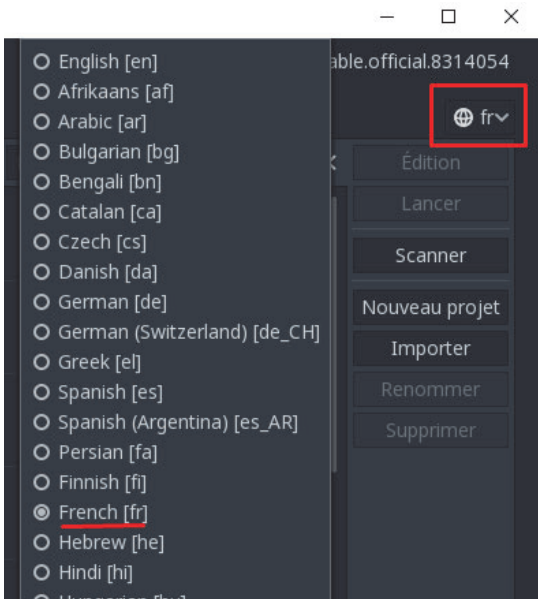
1.2. L'interface de Godot

Comme la plupart des moteurs de jeux évolués, Godot propose une interface de développement complète. Via cette interface, vous pourrez gérer vos ressources comme les modèles 3D, les scripts, les textures, les shaders, les musiques, etc. Vous pourrez également construire vos niveaux de façon visuelle via un éditeur WYSIWYG[1]. L'interface vous permet d'écrire vos scripts et de tester votre jeu à la volée directement dans l'éditeur.

Le gestionnaire de projets

La première fenêtre qui s'ouvre lorsque vous lancez Godot est le gestionnaire de projets. Cette fenêtre permet de créer des projets ou d'ouvrir un projet existant. En haut à droite, vous pourrez choisir la langue de l'interface.

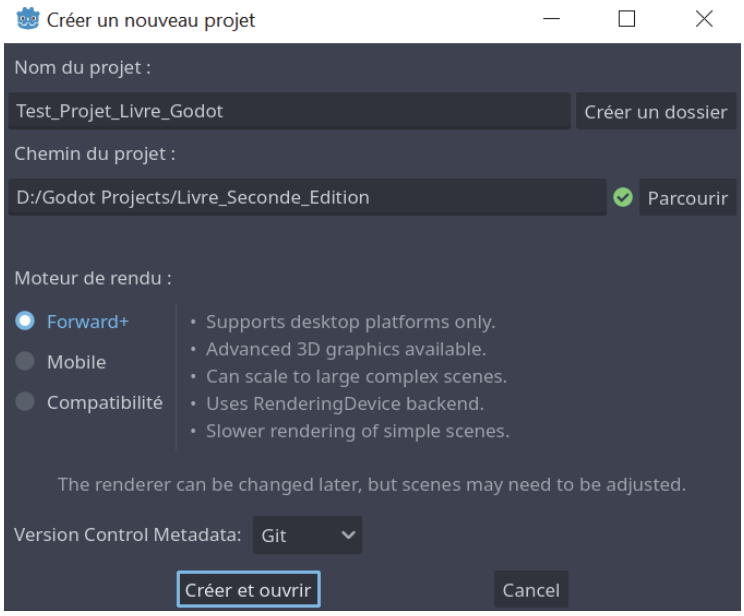
Figure 1.4 : *Choix de la langue*



Pour créer un nouveau projet, cliquez sur le bouton NOUVEAU PROJET. Ici, vous devrez spécifier un nom pour votre projet et un dossier de destination dans lequel tous les fichiers seront sauvegardés.

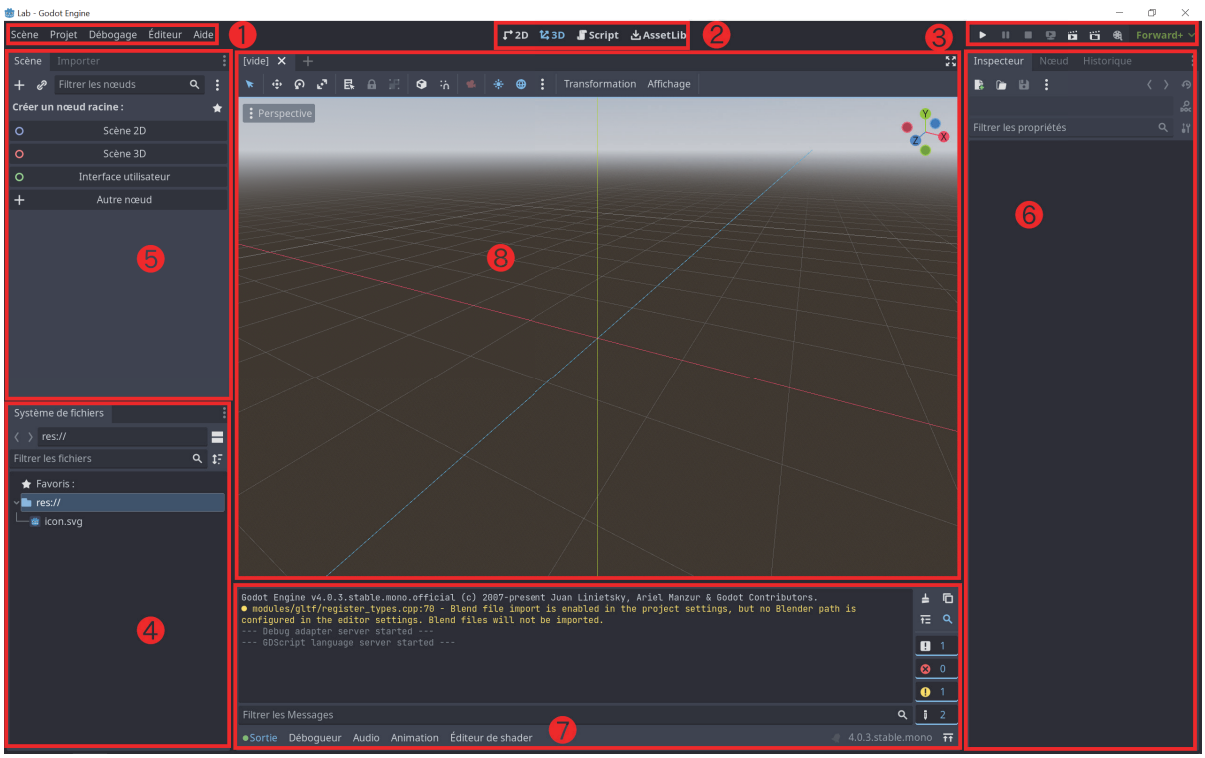
[1] *What You See Is What You Get*, c'est-à-dire *Ce que vous voyez est ce que vous obtenez*. Il s'agit d'interfaces permettant de créer de façon visuelle.

Figure 1.5 : *Création d'un nouveau projet*



Une fois les informations saisies, cliquez sur CRÉER ET OUVRIR pour lancer l'éditeur de Godot. Voici à quoi ressemble son interface et comment elle est organisée.

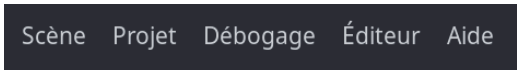
Figure 1.6 : *L'interface de Godot*



Comme vous pouvez le voir, plusieurs zones se distinguent. Examinons d'un peu plus près à quoi ces zones correspondent.

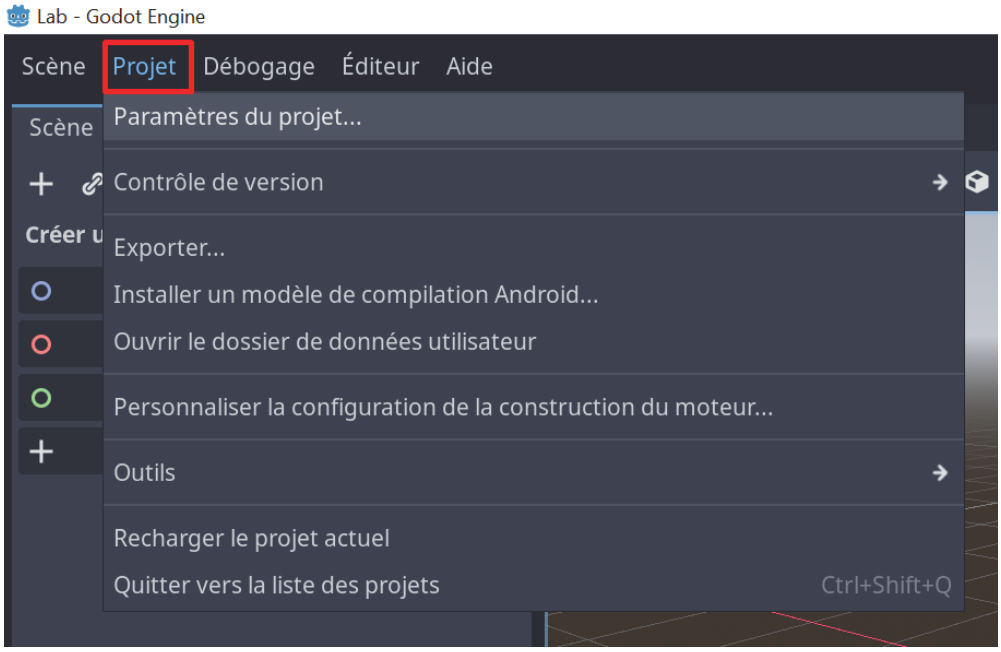
Le menu principal [❶]

Figure 1.7 : *Menu principal*



Le menu principal est très classique : il vous permet de créer une nouvelle scène, de sauvegarder votre scène, d'importer des fichiers, d'accéder aux paramètres du projet, de modifier l'affichage ou encore d'accéder à l'aide du logiciel. Dans notre cas, nous utiliserons principalement ce menu pour accéder aux options du projet via PROJET/PARAMÈTRES DU PROJET. Nous y reviendrons plus tard dans ce livre lorsque nous créerons notre premier jeu.

Figure 1.8 : *Utilisation du menu*



Les espaces de travail [❷]

Figure 1.9 : *Workspaces*

En haut au centre de l'éditeur, vous trouverez les workspaces. Ce menu vous permet d'accéder à un espace de travail. Par défaut nous sommes en 3D, cela permet de travailler dans l'éditeur de niveaux 3D. Vous pouvez passer en 2D pour créer des jeux 2D ou des menus ou encore passer en mode Script pour la programmation. Le dernier bouton permet d'accéder au magasin de ressources où vous pourrez trouver des éléments préconfigurés. Dans un premier temps, nous travaillerons en 2D.

Le menu de playtest [❸]

Playtester un jeu consiste à tester une partie de son jeu pour vérifier que le gameplay est fun. Comme vous pouvez le voir, j'utilise beaucoup de mots anglais. Dans le milieu du développement de jeux, certains mots ne se traduisent pas et s'utilisent directement en anglais. Vous trouverez les boutons de playtest en haut à droite de l'interface. Ils permettent de lancer le jeu, de le mettre en pause ou de l'arrêter.

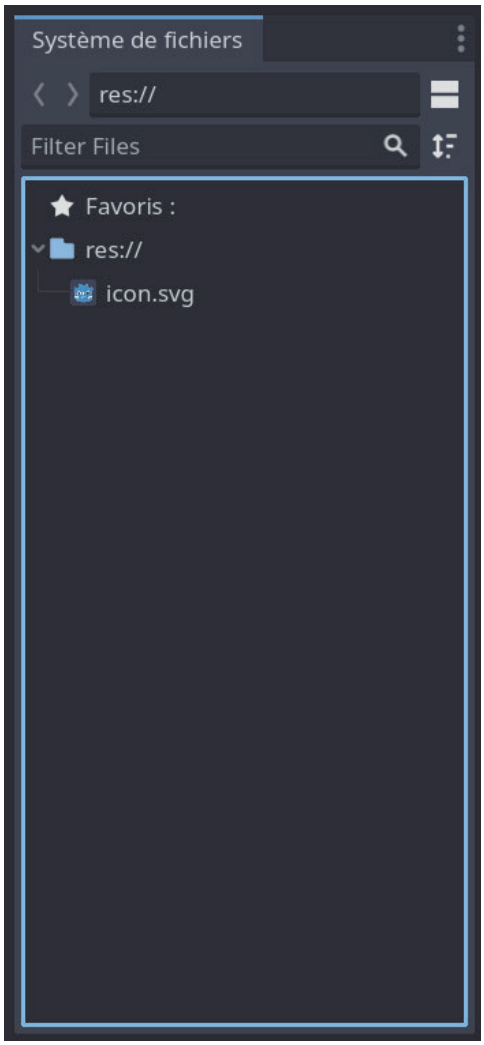
Figure 1.10 : *Playtest*

Vous pouvez lancer le jeu à l'aide du bouton PLAY ou via la touche F5. Pour tester votre jeu, vous devez spécifier une scène de démarrage, par exemple son menu principal.

Vous pouvez également tester la scène sur laquelle vous êtes en train de travailler. Pour cela, utilisez la cinquième icône ou le raccourci F6.

Le système de fichiers [❹]

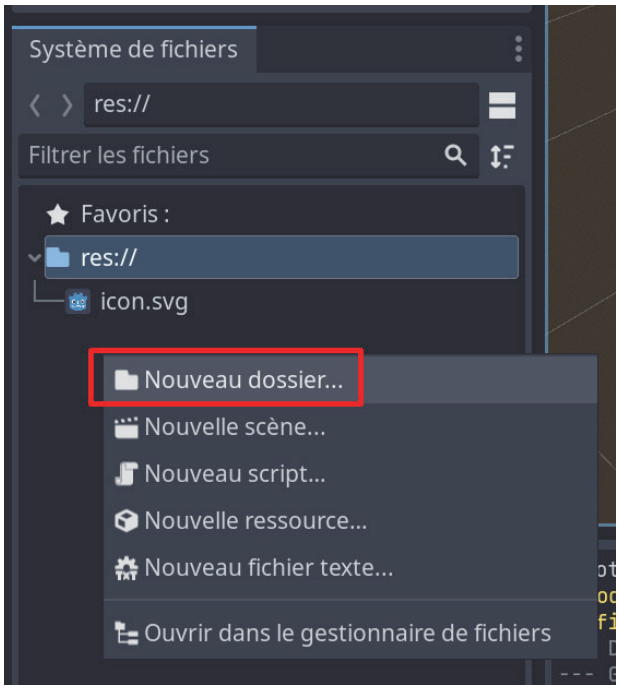
Figure 1.11 : *Système de fichiers*



Le système de fichiers liste l'ensemble de vos ressources (textures, sons, scènes, scripts, modèles 3D...) importées. Dès lors qu'un fichier (appelé *asset*) est importé, il est utilisable dans votre projet pour concevoir votre jeu.

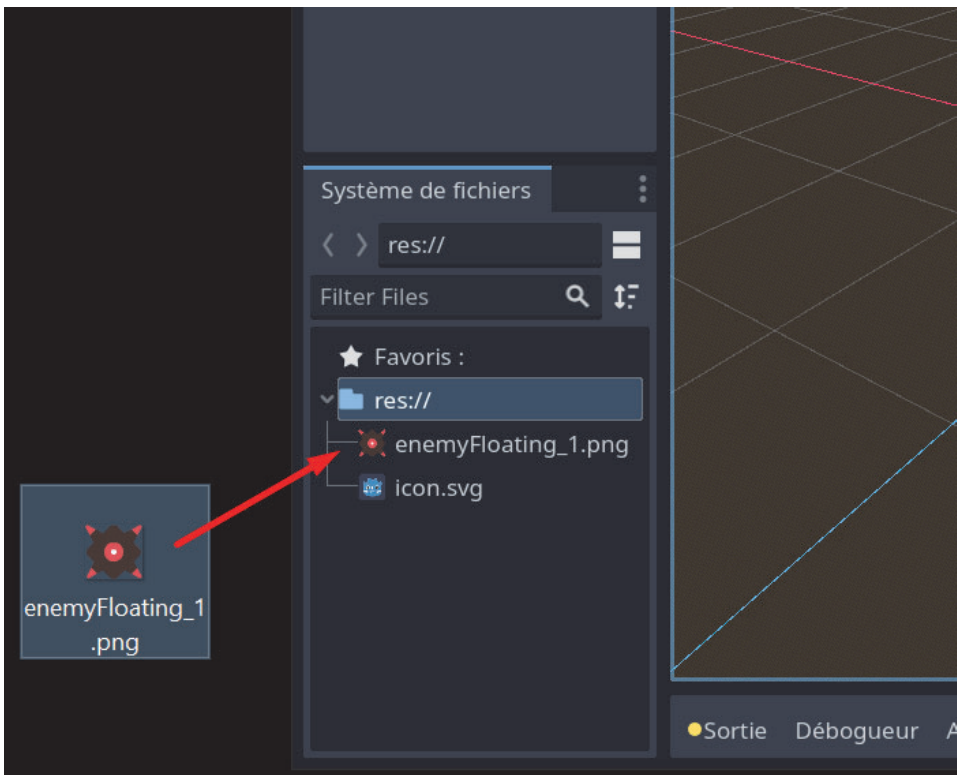
Pour créer des dossiers (et vous organiser), cliquez droit et sélectionnez NOUVEAU DOSSIER.

Figure 1.12 : *Création d'un dossier*



Pour importer des ressources comme des textures vous pouvez simplement les glisser/déposer dans la liste de vos assets ou dans un dossier.

Figure 1.13 : *Import d'assets*



Note › *L'image importée n'est qu'un exemple pour illustrer mes propos, il n'est pas nécessaire d'utiliser les mêmes ressources que moi dans ce chapitre. Ici nous nous contentons de découvrir le fonctionnement de base du logiciel.*

Lorsque nous créerons notre premier jeu, nous veillerons à garder une arborescence claire et propre. Nous rangerons toutes nos ressources dans des dossiers organisés.

La scène [❺]

Figure 1.14 : *La scène*

En haut à gauche de l'interface, vous avez la scène. C'est ici que vous retrouverez tous les objets que vous avez utilisés dans votre niveau. Par défaut la scène est vide et des raccourcis sont présents pour vous permettre de créer votre premier élément. Via cette fenêtre, vous pourrez organiser votre scène et accéder aux propriétés d'un objet en cliquant dessus. Les propriétés s'affichent dans l'inspecteur.

L'inspecteur (*inspector*) [❻]

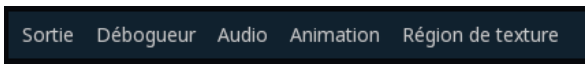
Figure 1.15 : *L'inspecteur*

L'inspecteur vous permet d'accéder à toutes les propriétés de l'objet sélectionné. Vous pourrez modifier les propriétés visuelles, les informations de position, rotation, taille et également les animations. Pour l'exemple j'affiche sur ma capture les propriétés d'une image.

Panneau inférieur [❼]

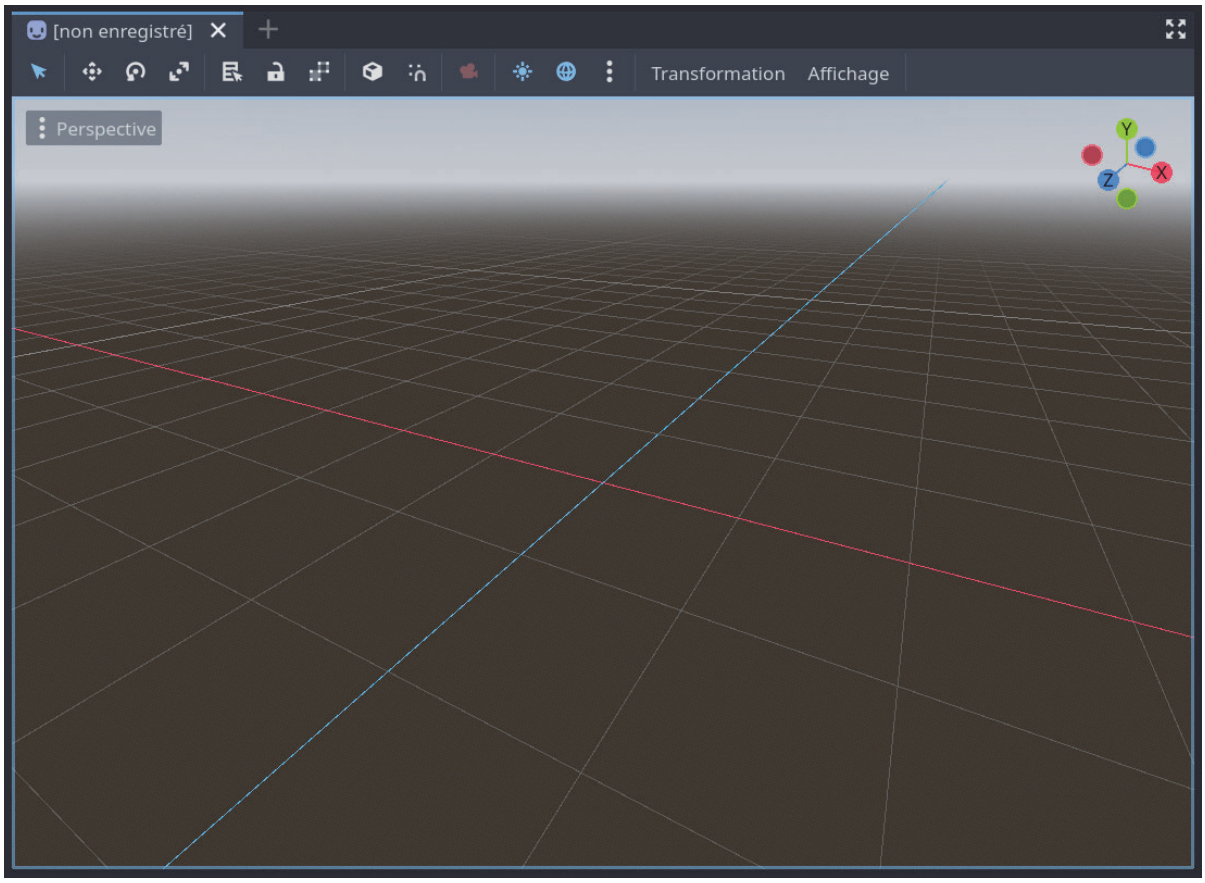
Tout en bas de l'écran se trouve le panneau inférieur. Refermé par défaut, il vous permet d'accéder à de nombreux outils comme le système de mixage audio, le système d'animation mais également au débogueur et à la console qui affichera les données du code et les erreurs.

Figure 1.16 : *Panneau inférieur*



Fenêtre principale (*viewport*) [❽]

Figure 1.17 : *Viewport*



Au centre de l'interface se trouve le viewport, la fenêtre principale de Godot. C'est ici que nous travaillerons, que nous positionnerons les éléments décoratifs, les per-

sonnages, afin de créer les différents niveaux de notre jeu. Nous pouvons travailler en 2D et en 3D.

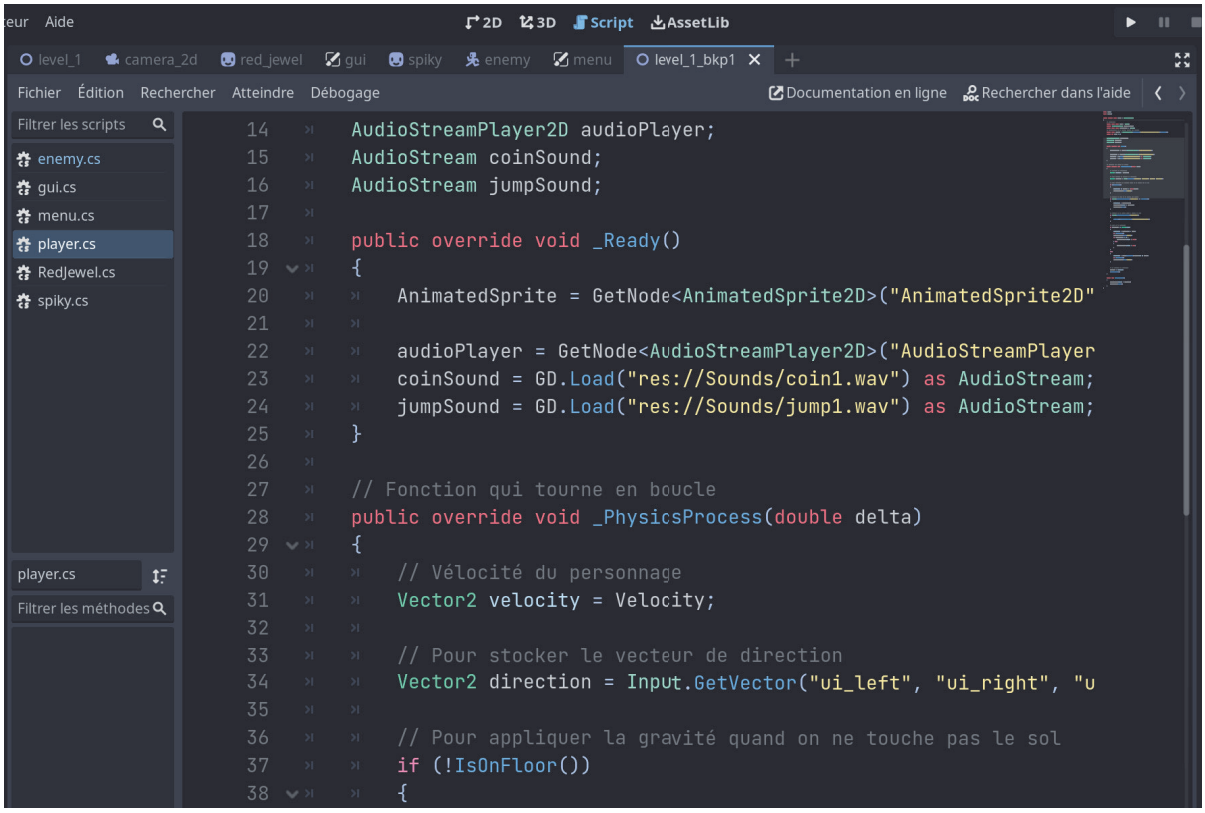
Vous pouvez naviguer dans cette fenêtre avec le clavier et la souris. Avec un clic droit, vous pouvez tourner la caméra (en mode 3D). En mode 2D, appuyez sur le bouton du milieu (la molette) pour déplacer la caméra ; et en mode 3D, maintenez la touche Maj enfoncée pendant que vous appuyez sur la molette.

La molette permet aussi de zoomer/dézoomer.

En haut de cette fenêtre, vous trouverez une barre d'outils permettant d'accéder aux outils de sélection, position, rotation et modification de taille.

C'est également sur cet espace que nous pourrons écrire du code.

Figure 1.18 : *Éditeur de code*



Nous verrons dans la suite de ce livre d'autres zones de l'interface au fur et à mesure de nos besoins.

1.3. Modification de l'interface

Vous pouvez adapter l'interface à vos habitudes. Les fenêtres peuvent être redimensionnées en attrapant la petite arête entre deux fenêtres.

Figure 1.19 : *Agencer l'interface*

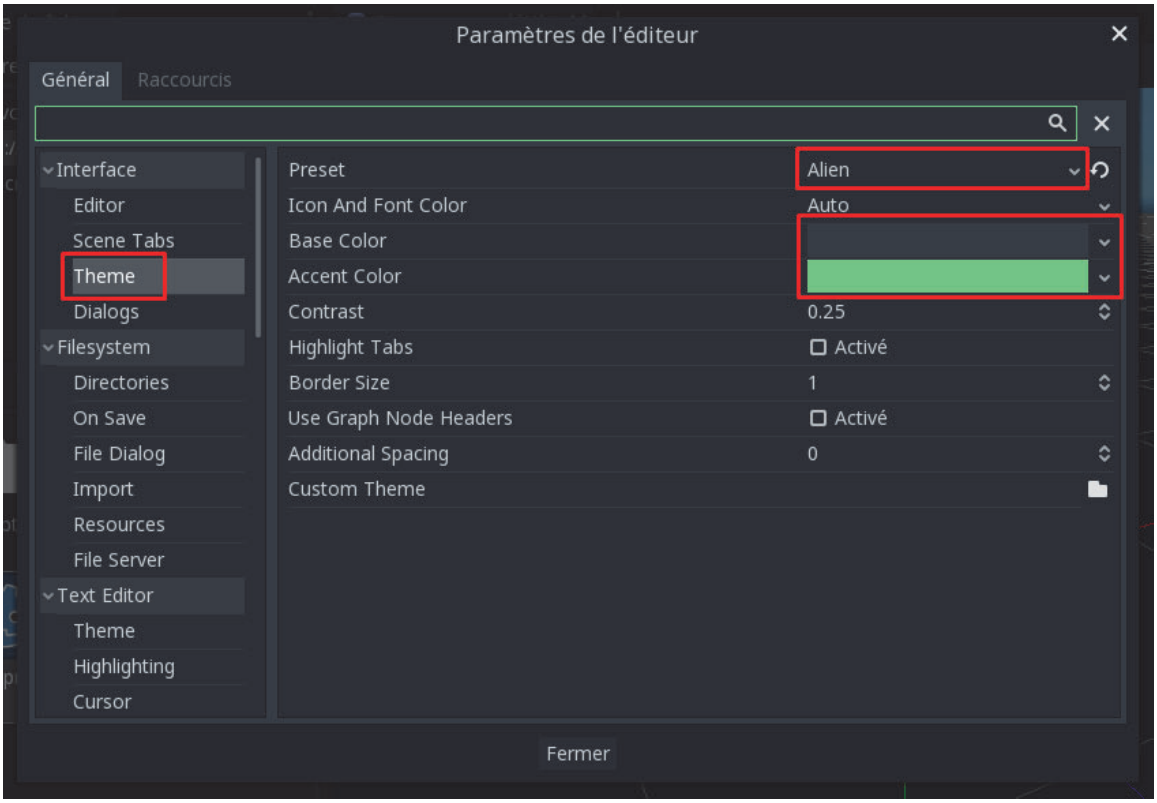
Vous pouvez aussi repositionner les éléments et les ancrer où vous le souhaitez. Pour cela, cliquez sur les points en haut à droite des fenêtres pour accéder à l'outil de position :

Figure 1.20 : *Position du dock*

Pour ma part, je travaille avec le système de fichiers et la scène à gauche, l'inspecteur à droite et le viewport au centre.

Vous avez également la possibilité de modifier les couleurs et le thème de Godot. Pour cela, cliquez sur ÉDITEUR/PARAMÈTRES DE L'ÉDITEUR puis cliquez sur THEME et choisissez les couleurs ou le préréglage (*preset*) de votre choix.

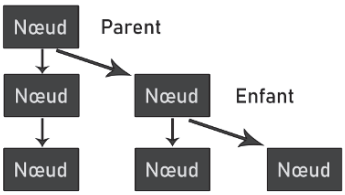
Figure 1.21 : *Modification du thème*



1.4. Nœuds et arbres

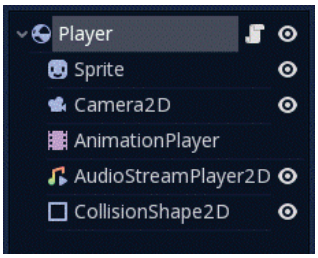
Godot est basé sur un système de nœuds [nodes] et d'arbres [tree]. Les nœuds sont les éléments fondamentaux pour la création de jeux. Ils ont un nom, des propriétés et peuvent être organisés afin d'avoir des enfants comme sur la Figure 1.22.

Figure 1.22 : *Un arbre composé de nœuds*



Lorsqu'ils sont organisés de cette façon, nous appelons cela un arbre (*tree*). Avec Godot vous aurez la possibilité de créer des nœuds, de les organiser, de les assembler afin de créer des jeux élaborés. Voici par exemple à quoi pourrait ressembler un nœud joueur :

Figure 1.23 : *Exemple de nœud joueur*



Comme vous pouvez le voir, le joueur a un sprite pour son visuel, une caméra pour que le joueur puisse le voir, un système d'animation, un système de son et un objet de collision. C'est typiquement le genre de nœud que nous aurons l'occasion de créer.

Dans le prochain chapitre nous verrons plus en détail comment se déroule la création d'une scène sous Godot.

2

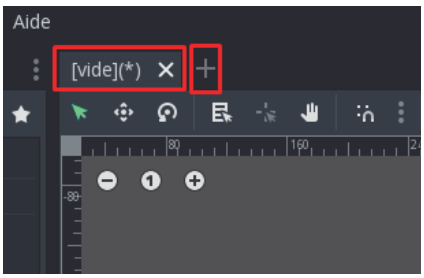
Création d'une scène sous Godot

Afin de prendre en main l'interface de Godot et de bien comprendre son fonctionnement, nous allons créer une scène basique et voir comment se déroule ce processus.

Une scène est composée de nœuds organisés sous forme d'arbres. Une scène a un seul nœud racine. Les scènes peuvent être sauvegardées pour être modifiées par la suite et assemblées pour créer un jeu à plusieurs niveaux. En général, une scène ne correspond pas à un niveau mais plutôt à un élément de jeu comme un personnage, le décor d'arrière-plan, l'interface utilisateur, etc. Plusieurs scènes sont ensuite assemblées pour former un niveau.

Si cela n'est pas déjà fait, créez un nouveau projet comme nous l'avons vu au chapitre précédent. Ce projet nous permettra de faire nos quelques tests. Par défaut, lorsque vous créez un nouveau projet, une scène vide apparaît.

Figure 2.1 : *Nouvelle scène vide*

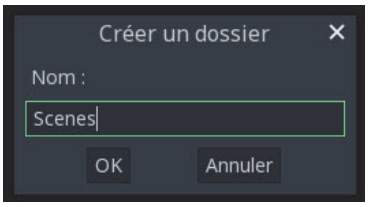


Le petit + qui apparaît à droite de l'onglet de la scène permet de créer une nouvelle scène. Pour le moment, nous allons travailler sur la scène par défaut.

2.1. Organisation du projet

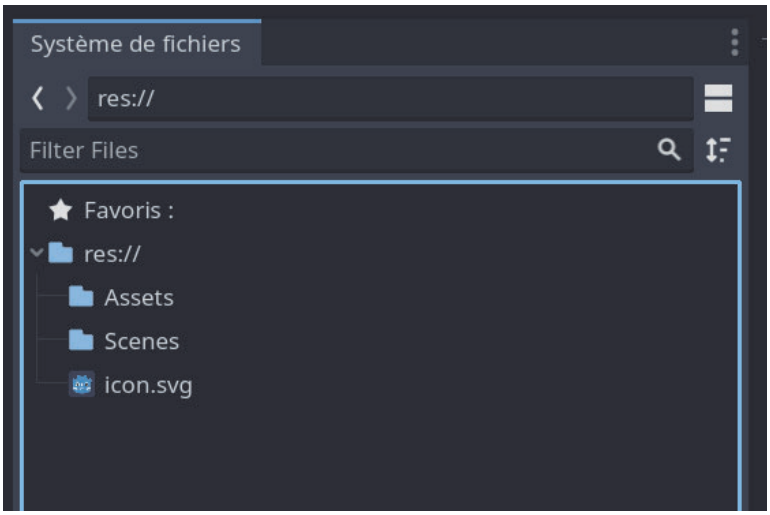
Pour notre premier essai, nous allons nous amuser à créer une balle rebondissante. Une bonne pratique consiste à créer une scène par objet réutilisable. Un personnage est réutilisable, une plateforme est réutilisable, une caméra est réutilisable... Tous ces objets seront donc organisés dans des scènes. Nous allons commencer par créer un nouveau dossier `Scenes` dans lequel nos scènes seront enregistrées. Faites un clic droit dans le système de fichiers et choisissez NOUVEAU DOSSIER pour créer ce dossier.

Figure 2.2 : *Création d'un dossier*



Vous pouvez également créer un dossier `Assets` dans lequel nous importerons nos images. Votre architecture de dossiers devrait ressembler à cela :

Figure 2.3 : *Notre arborescence*



2.2. Import des textures

Nous allons ensuite importer une texture qui représentera notre balle rebondissante. Pour cela, nous allons commencer par rechercher une texture de balle 2D gratuite sur internet. Je vous conseille le site OpenGameArt qui propose de nombreuses ressources gratuites et open-source. Vous y retrouverez par exemple, en recherchant `Ball`, une texture comme celle de la Figure 2.4.

Figure 2.4 : *Une texture gratuite*

C'est cette texture que je vais télécharger. Si vous le souhaitez, vous pouvez rechercher des alternatives sur un site d'icônes comme Iconfinder.

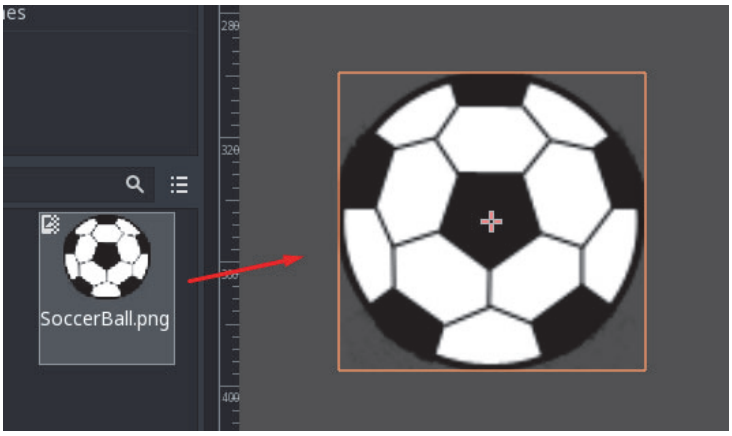
Figure 2.5 : *D'autres textures gratuites*

Une fois que vous avez votre texture, glissez/déposez-la dans Godot dans le dossier `Assets`.

Figure 2.6 : *Import de la balle*

Pour ajouter une texture à votre scène, il vous suffira de la glisser/déposer du système de fichiers vers le viewport. Une fois ceci fait, votre image apparaîtra.

Figure 2.7 : *Utilisation d'une texture*

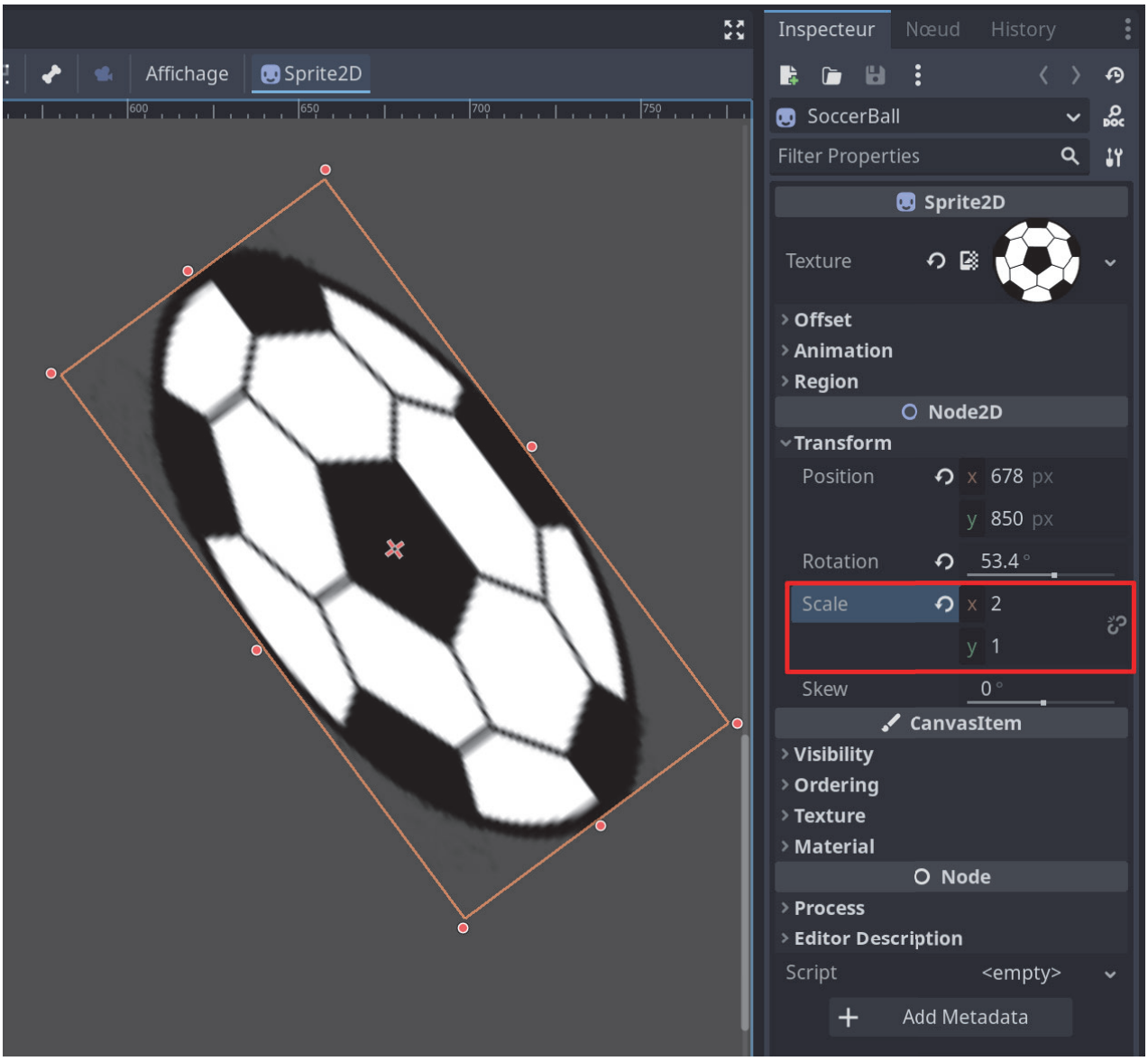


Votre texture est alors ajoutée. Vous pouvez la déplacer avec la souris mais également la tourner avec l'outil de rotation.

Figure 2.8 : *Outil de rotation*

Si vous le souhaitez, vous pouvez examiner les propriétés de votre texture dans l'inspecteur. Si votre texture est bien sélectionnée, vous y trouverez de nombreux paramètres. Un paramètre très important est le TRANSFORM de votre objet. Le TRANSFORM permet d'accéder à sa position, sa rotation et sa taille. Par exemple, si vous changez la propriété SCALE en indiquant la valeur 2 en X, votre ballon sera aplati.

Figure 2.9 : *Modification du Scale*



Faites Ctrl+Z pour revenir en arrière.

2.3. Création d'une balle physique

Notre balle doit être rebondissante : cela signifie qu'elle doit être affectée par la gravité, gérer les collisions et rebondir lorsqu'elle touche le sol. Nous allons donc créer un nœud pour notre balle qui respectera ces propriétés.

Commençons par supprimer notre balle précédente. Pour cela, dans la fenêtre Scène, cliquez droit sur votre balle et choisissez SUPPRIMER NŒUD. De cette façon nous allons repartir sur une scène vide.

Lorsque nous créons des nœuds, il y a un certain ordre à respecter. Par exemple, dans notre cas, le nœud principal (*root node*) sera un RigidBody. Le Sprite, notre image, sera enfant du RigidBody. Nous verrons dans un instant pourquoi l'ordre est important. Commençons par créer notre RigidBody 2D.

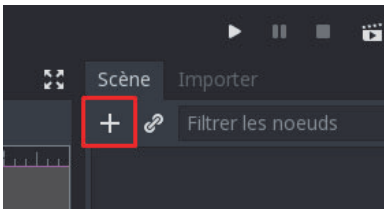
RIGIDBODY

Le *RigidBody* est le composant permettant de faire en sorte que la gravité soit appliquée à un objet de jeu. Le RigidBody possède de nombreuses propriétés qui peuvent être adaptées pour que le résultat soit réaliste. Vous pouvez modifier la masse, le poids, les propriétés de friction et de nombreux autres paramètres. Le RigidBody est décliné en deux versions : la version standard et la version 2D appelée le *RigidBody2D*. Lorsque vous créez un nœud comme un RigidBody2D, une alerte s'affiche et vous indique quels nœuds enfants sont nécessaires au bon fonctionnement de ce nœud parent.

Le RigidBody est adapté aux objets physiques comme les balles. Il existe d'autres types de composants physiques. Vous retrouverez le *CharacterBody* (ou *CharacterBody2D*) qui est plus adapté aux personnages ou encore les *StaticBody* (ou *StaticBody2D*) adaptés aux éléments du décor solides mais qui ne bougent jamais comme une plateforme. Nous aurons l'occasion de manipuler ces différents composants.

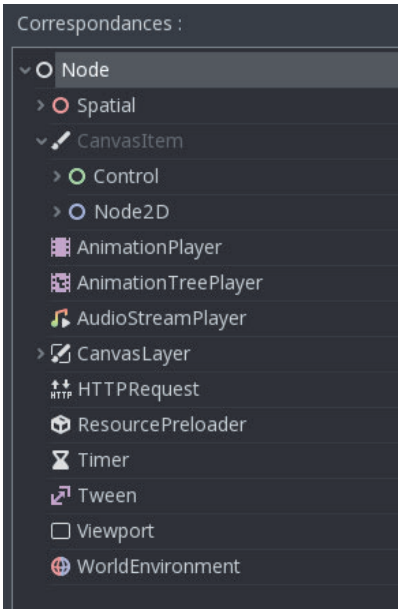
Pour créer un nœud, cliquez sur le + se trouvant au niveau de votre scène :

Figure 2.10 : *Création d'un nœud*



Une fenêtre s'ouvre alors. Dans cette fenêtre, vous retrouverez tous les composants qui peuvent être utilisés en tant que nœud.

Figure 2.11 : *Liste des nœuds*



La liste est longue et vous pourrez utiliser la barre de recherche pour vous y retrouver plus facilement. Il y a quelques grandes catégories de nœuds que je détaille dans l'encadré ci-dessous.

DIFFÉRENTS TYPES DE NŒUDS [NODES]

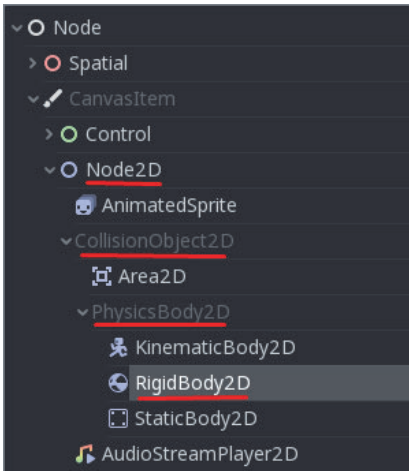
Les nœuds en rouge (comme `Spatial` et ses enfants) sont spécifiques à la 3D. Les nœuds en bleu (comme `Node2D` et ses enfants) sont quant à eux dédiés à la 2D. Les nœuds verts (comme `Control`) sont destinés à tous les éléments d'interface comme les menus et interfaces utilisateurs.

Les nœuds `Node3D` ou `Node2D` sont des nœuds vides. Ils peuvent être utilisés comme un nœud principal qui aura des enfants. Cela peut être pratique pour mieux s'organiser.

D'autres types de nœuds existent comme des nœuds spécifiques aux animations, aux sons, au temps... Nous y reviendrons plus tard.

Dans notre cas, nous recherchons le RigidBody2D. Ce nœud se trouve dans `Node2D/CollisionObject2D/PhysicsBody2D/RigidBody2D`.

Figure 2.12 : *Nœud RigidBody2D*



Vous pouvez également rechercher `RigidBody2D` dans la barre de recherche pour aller plus vite. Si vous savez ce que vous recherchez, la barre de recherche est plus pratique. Sélectionnez le composant RigidBody2D puis cliquez sur le bouton CRÉER. Le nœud est alors généré.

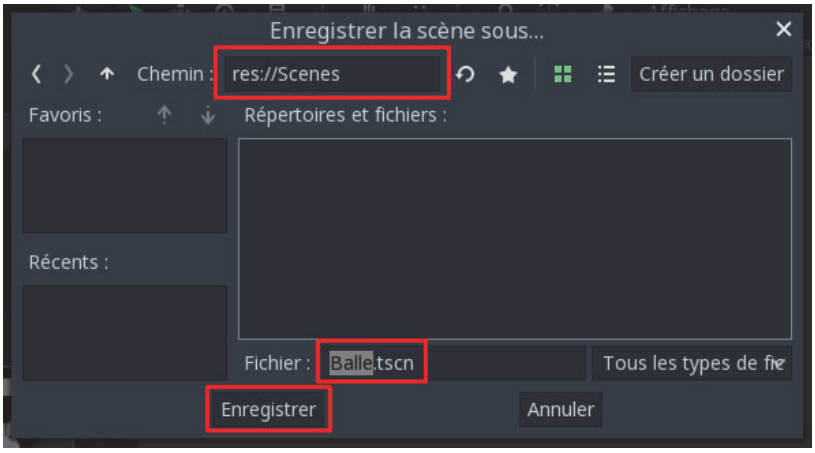
Figure 2.13 : *Création du nœud*

Double-cliquez sur ce nœud pour le renommer et saisissez `Balle`.

Figure 2.14 : *Renommer un nœud*

Sauvegardez votre scène pour ne pas perdre votre travail. Pour cela, faites un Ctrl+S et rendez-vous dans le dossier `Scenes` pour enregistrer la scène `Balle.tscn`.

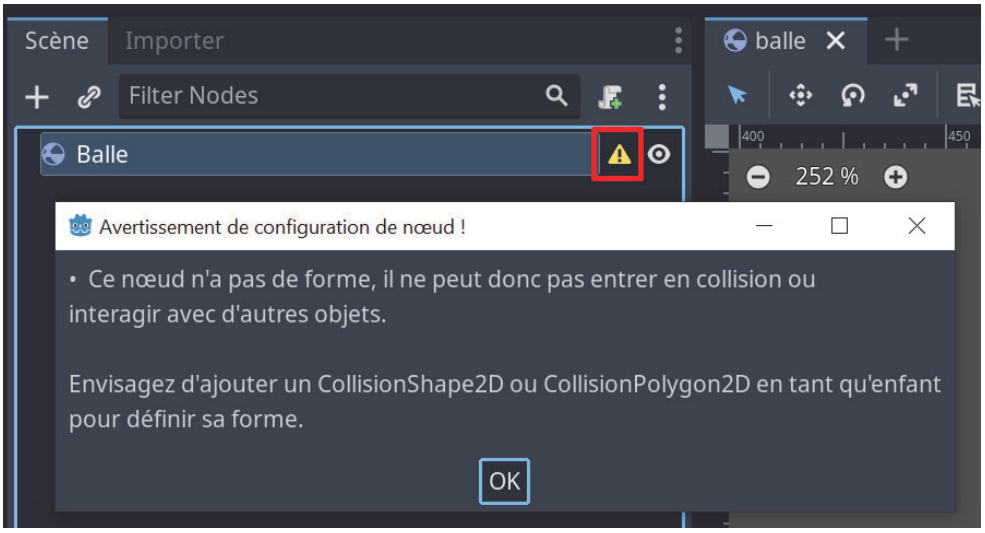
Figure 2.15 : *Sauvegarder une scène*



Maintenant votre travail est sauvegardé sous forme de scène. Vous pourrez éditer cette scène et l'utiliser pour votre futur jeu.

Vous l'aurez probablement remarqué mais votre RigidBody2D est marqué d'un triangle jaune WARNING. Cette alerte indique que vous devez ajouter un composant de collision.

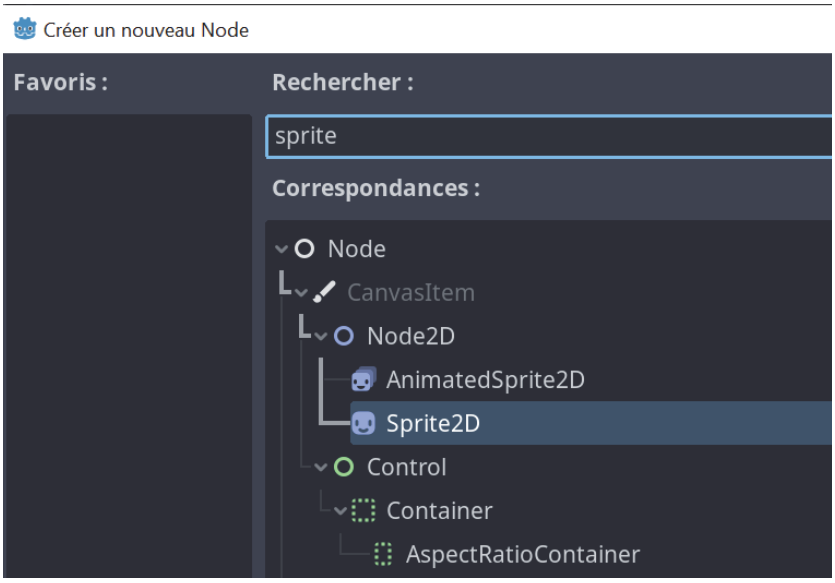
Figure 2.16 : *Avertissement*



Nous allons donc remédier à cela. Avant toute chose, nous allons ajouter un visuel à notre objet afin qu'il soit visible. Cliquez sur votre RigidBody2D et appuyez de nouveau

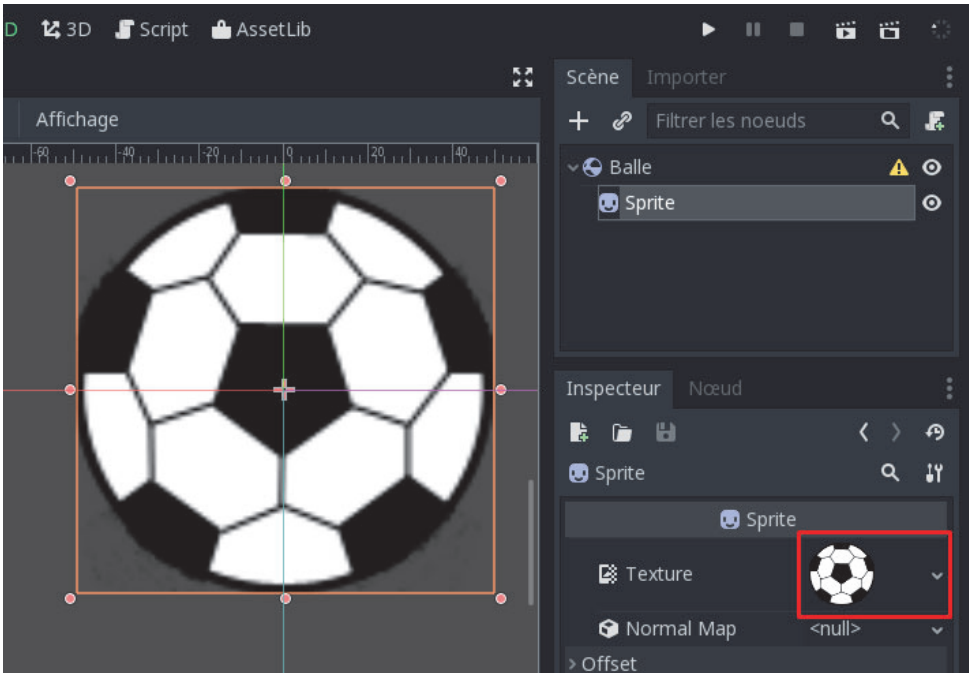
sur le + pour ajouter un nœud. Le raccourci clavier pour ajouter un nœud est Ctrl+A. Recherchez `Sprite2D`, sélectionnez ce composant et cliquez sur CRÉER :

Figure 2.17 : *Création d'un Sprite2D*



Pour le moment notre Sprite est vide. Nous devons ajouter une texture pour lui attribuer un visuel. Avec votre Sprite sélectionné, glissez/déposez la texture du ballon dans la propriété TEXTURE de votre Sprite.

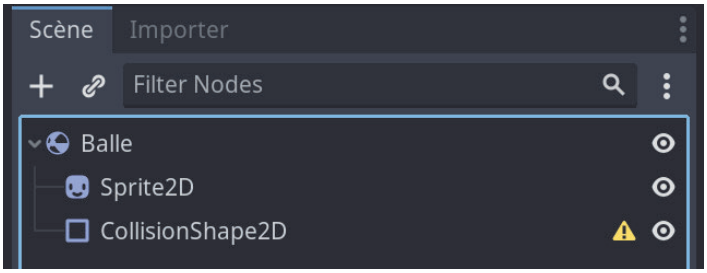
Figure 2.18 : *Ajout d'une texture*



Nous allons ensuite ajouter le fameux composant de collision : le CollisionShape2D. C'est ce composant qui vous permettra de détecter si l'objet est en collision avec un autre objet. Sélectionnez la `Balle` (Attention, je parle du RigidBody2D, pas du Sprite) et ajoutez un nœud. Ce nœud sera un `CollisionShape2D`.

Si vous ne vous êtes pas trompé, votre scène devrait ressembler à cela :

Figure 2.19 : *Ajout du CollisionShape2D*

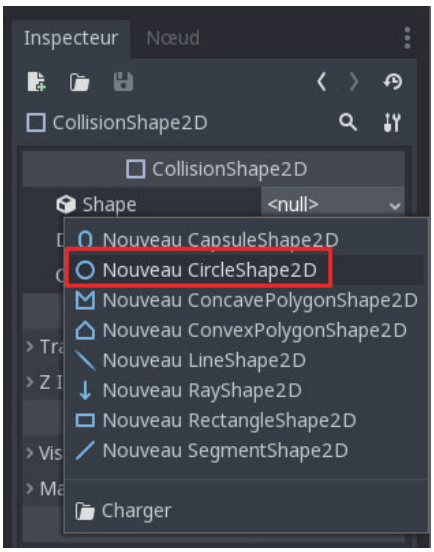


Attention *› Veillez à bien sélectionner le RigidBody et non votre Sprite sinon votre CollisionShape sera enfant de l'image et pas du RigidBody.*

Comme vous l'avez remarqué, le CollisionShape2D affiche une alerte. Il vous indique que vous devez créer une forme. Une forme (ou *Shape* en anglais) correspond à la forme qu'aura votre composant de collision. Vous pouvez créer un cercle, un carré, une capsule... Dans notre cas, nous allons créer un cercle.

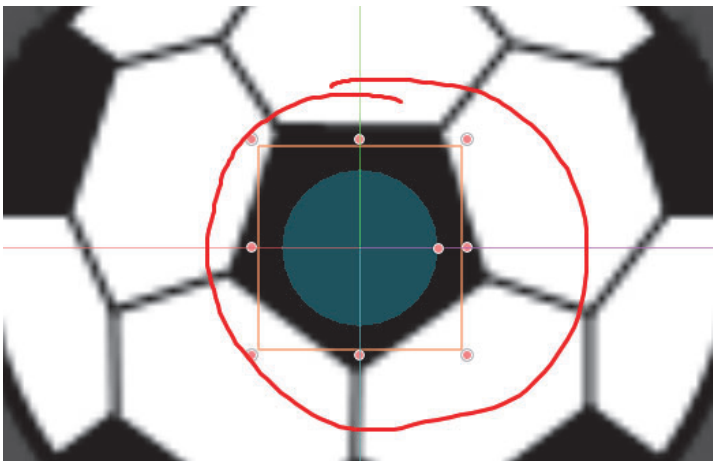
Cliquez sur la propriété SHAPE et sélectionnez `CircleShape`.

Figure 2.20 : *Ajout du CircleShape*



Vous remarquerez que votre composant de collision apparaît sur votre balle.

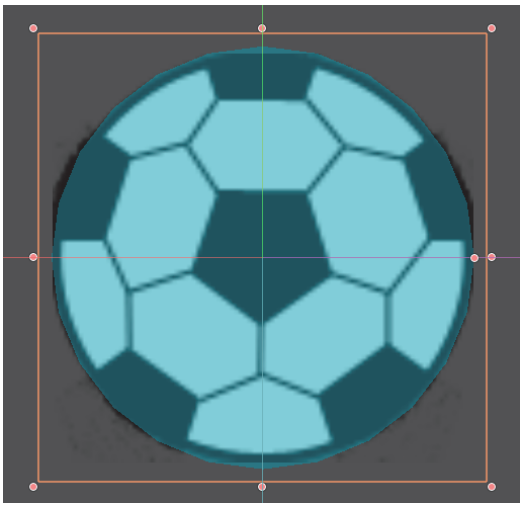
Figure 2.21 : *Le CollisionShape*



Pour le moment, le Shape est trop petit par rapport à notre image. Nous allons le redimensionner. Utilisez la petite puce de redimensionnement pour cela.

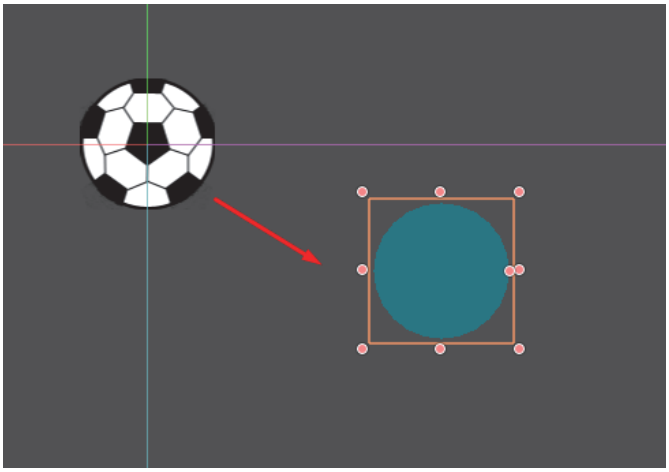
Agrandissez votre Shape pour englober votre balle :

Figure 2.22 : *Modification de la taille du Shape*



Nous allons maintenant positionner notre balle au centre de l'écran. Si vous essayez de déplacer la balle avec la souris, vous risquez d'avoir la surprise suivante :

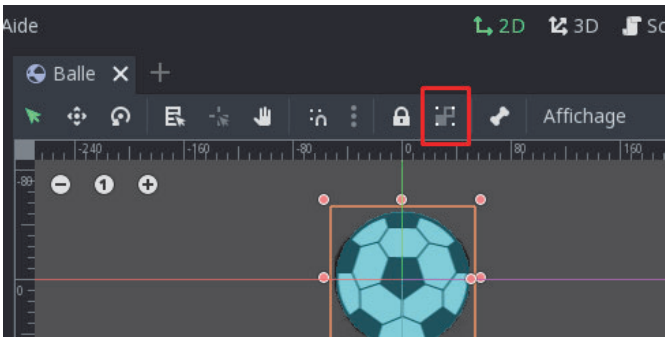
Figure 2.23 : *Tentative de déplacement de la balle*



En essayant de déplacer la balle, nous déplaçons l'objet de collision qui se trouve au-dessus la balle. Ce que nous souhaitons, c'est tout déplacer en même temps comme un seul objet. Pour cela, Godot propose un outil permettant de verrouiller les enfants du nœud. Cliquez sur le RigidBody2D, qui est le nœud principal, et cliquez sur l'icône RENDRE LA SÉLECTION DES ENFANTS IMPOSSIBLE de la barre d'outils.

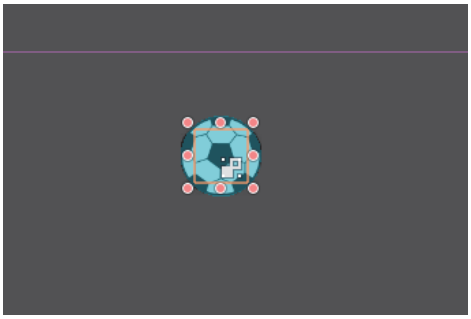
Note *› Selon les versions, cette option peut aussi s'appeler GROUPER LES NŒUDS SÉLECTIONNÉS.*

Figure 2.24 : *Outil de verrouillage*



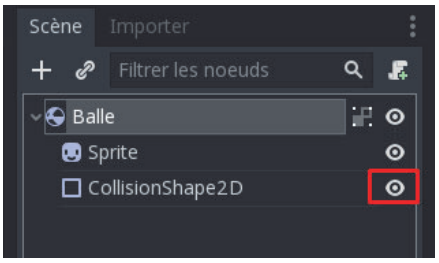
Cet outil permet de bloquer le déplacement des enfants. Maintenant, nous pouvons déplacer notre sphère convenablement :

Figure 2.25 : *La sphère centrée*



Enfin, petit détail, si le visuel du CollisionShape vous gêne, vous pouvez le masquer en cliquant sur l'œil dans la scène.

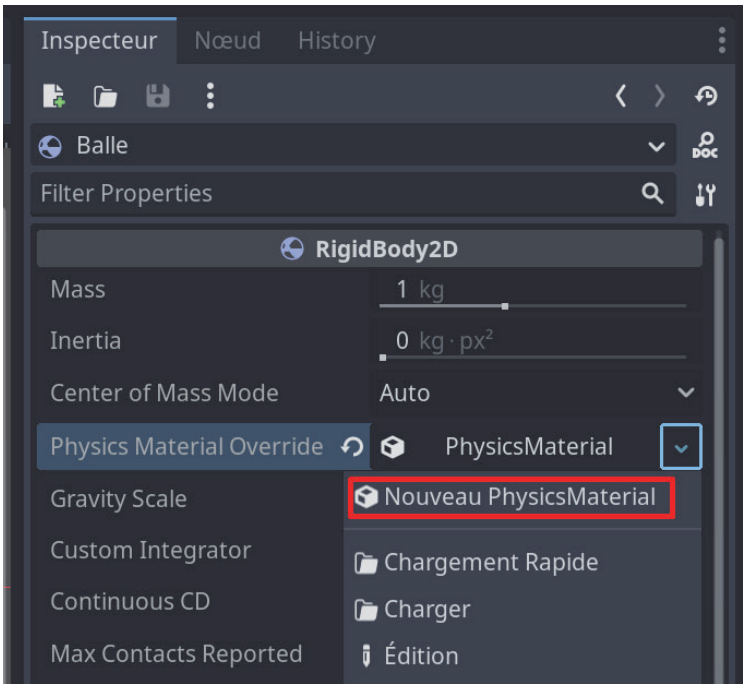
Figure 2.26 : *Masquer un nœud*



Pensez à enregistrer votre travail en utilisant Ctrl+S. Vous pouvez vérifier que tout fonctionne bien en appuyant sur la touche F6. Cette touche permet de lancer la scène en cours. Si tout est bien configuré votre projet devrait se lancer et la balle devrait tomber dans le vide à cause de la gravité.

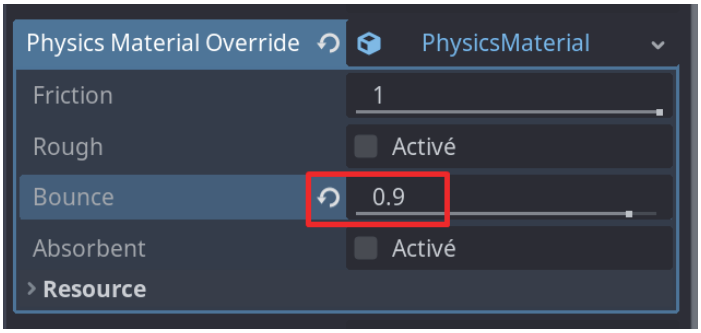
Notre objectif était de créer une balle rebondissante. Nous allons devoir configurer les propriétés physiques de notre objet et lui ajouter la faculté de rebondir (appelée *bounce*). Pour cela, il nous faut un PhysicsMaterial. Ajoutez-en un en sélectionnant NOUVEAU PHYSICSMATERIAL dans le menu déroulant associé à la propriété PHYSICSMATERIAL.

Figure 2.27 : *Ajout d'un PhysicsMaterial*



Il faut maintenant rendre ce PhysicsMaterial rebondissant. Cliquez dessus et augmentez la valeur de son paramètre BOUNCE (rebond). C'est ce dernier qui détermine sa capacité à rebondir. Ici j'ai mis 0.9. Si vous augmentez la valeur à 1, le rebond sera parfait et infini. Dans mon cas, le rebond diminuera lentement.

Figure 2.28 : *Modification de la propriété Bounce*



Vous avez maintenant une balle configurée. Enregistrez votre scène. Dans le prochain chapitre, nous verrons comment utiliser nos scènes grâce à l'instanciation.

3

L’instanciation avec Godot

Créer un jeu en créant une unique scène avec plusieurs nœuds est possible pour un petit projet, mais en réalité nous ne faisons jamais cela. Les projets évoluent, grandissent, et pour garder une architecture simple et propre, nous ne créons jamais de scène géante. Avec Godot, les éléments de votre jeu sont séparés en scènes. Le personnage est une scène, une plateforme est une scène, l’interface utilisateur et une scène... Toutes ces scènes sont ensuite importées dans une scène principale qui va les regrouper. Cela s’appelle l’*instanciation*. Pour comprendre ce concept et le mettre en pratique, nous allons créer une plateforme, qui constituera une scène, et nous créerons une scène principale dans laquelle nous instancierons cette plateforme et la balle rebondissante que nous avons déjà créée.

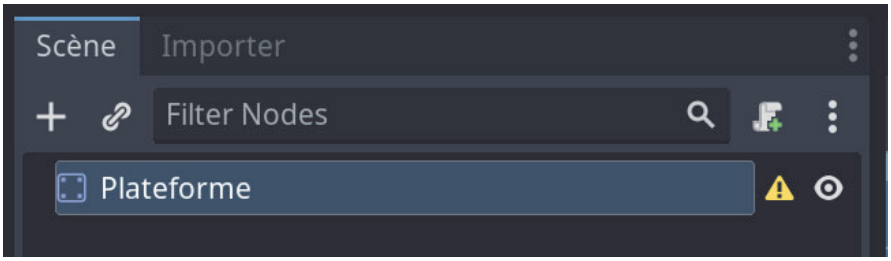
3.1. Création de la plateforme

Pour la création de la plateforme, nous irons un peu plus vite car la méthode est similaire à celle utilisée pour la balle. Commencez par créer une nouvelle scène en cliquant sur le + en haut du viewport ou via le menu SCÈNE/NOUVELLE SCÈNE. Recherchez une texture de plateforme sur internet :

Figure 3.1 : Recherche d'une plateforme 2D

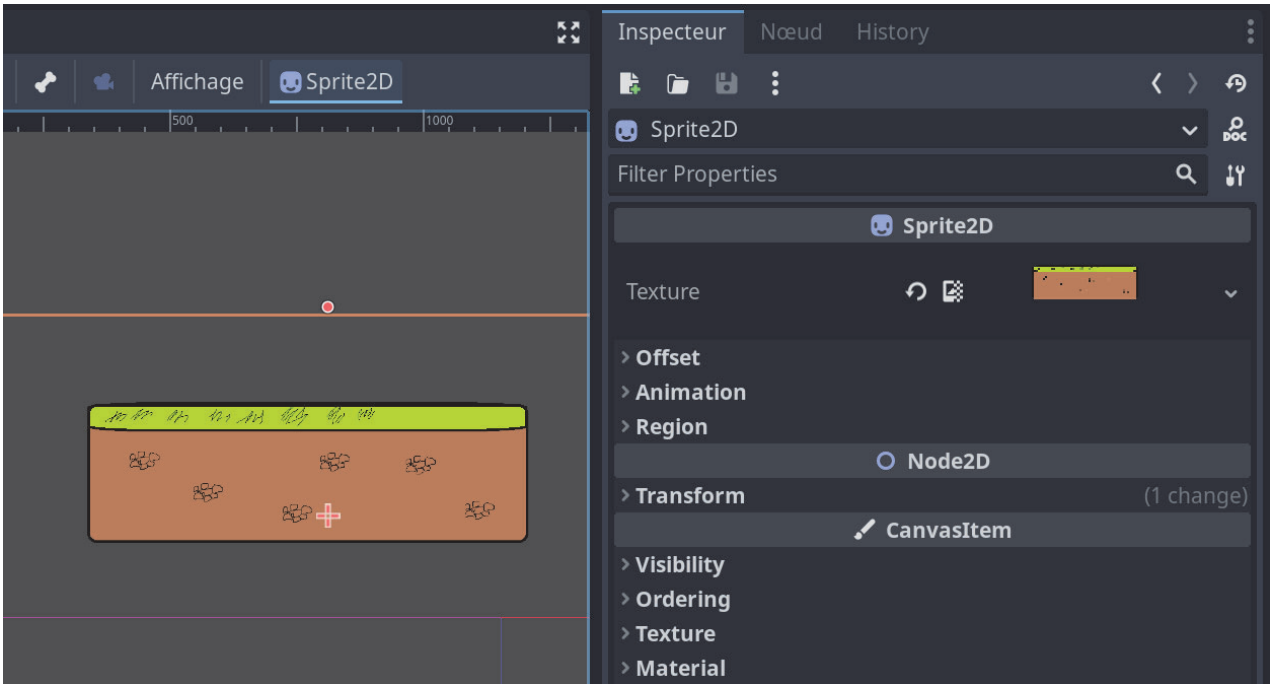
Une fois votre plateforme trouvée, importez-la dans le dossier `Assets`. Dans votre nouvelle scène, créez un nœud StaticBody2D et renommez-le `Plateforme`.

Figure 3.2 : *Création du nœud*



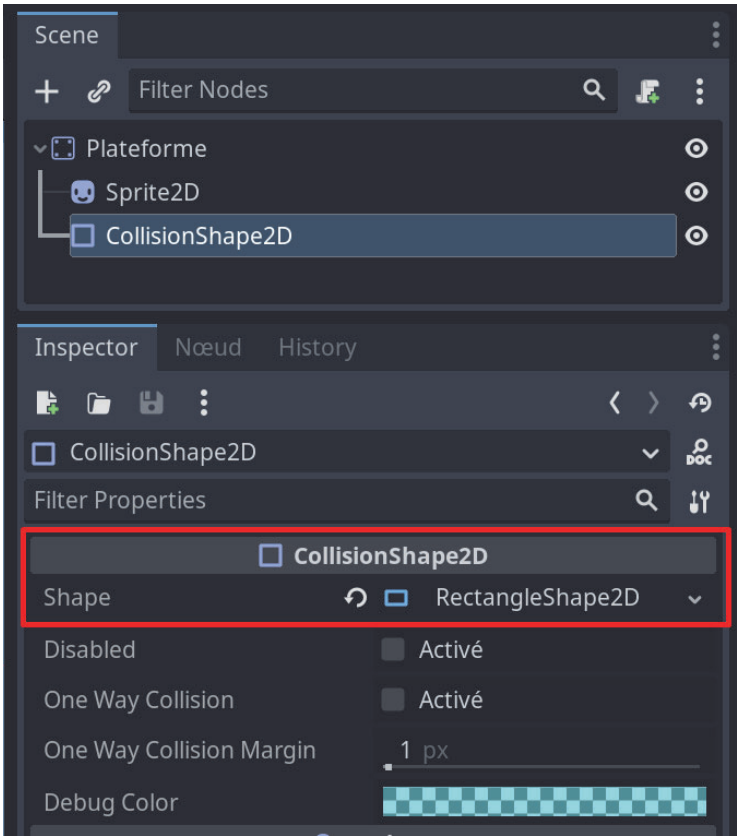
Ajoutez un Sprite2D à votre nœud afin de lui attribuer la texture de votre plateforme.

Figure 3.3 : *Ajout du Sprite2D*



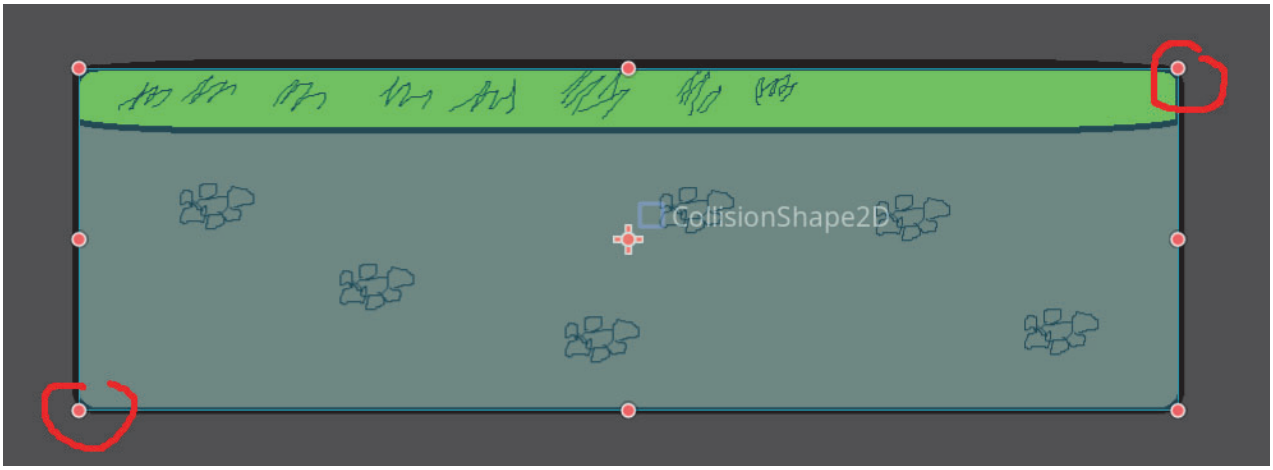
Vous devez également ajouter un CollisionShape2D à votre objet. Ajoutez-le en tant qu'enfant du StaticBody et choisissez un RectangleShape2D.

Figure 3.4 : *Ajout du CollisionShape2D*



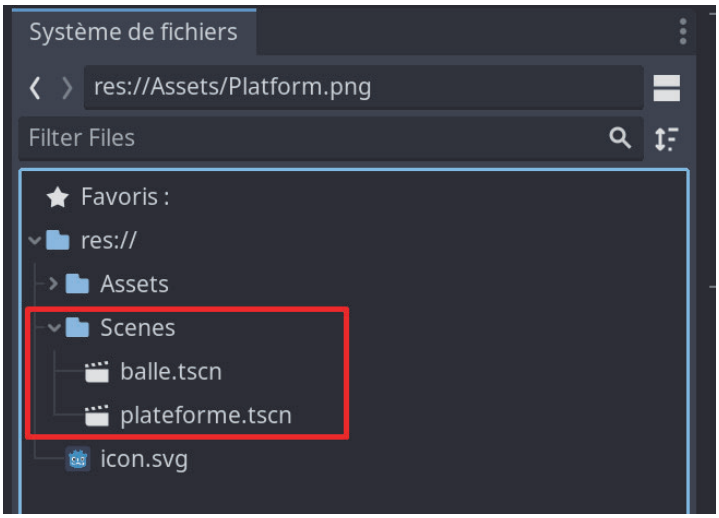
Utilisez les petites puces pour redimensionner votre Shape afin d'englober votre image comme sur la Figure 3.5.

Figure 3.5 : *Puces de redimensionnement du CollisionShape2D*



Pensez à cliquer sur votre StaticBody et à cocher l'option rendant impossible la sélection des enfants. Vous pouvez également masquer le CollisionShape. Une fois votre plateforme configurée, sauvegardez-la dans une scène. Vous devez maintenant avoir deux scènes distinctes, une pour la balle, l'autre pour la plateforme.

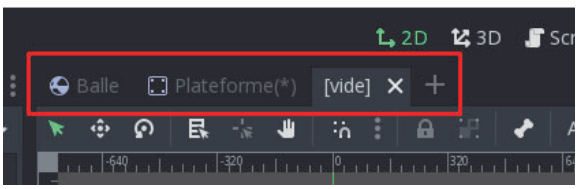
Figure 3.6 : *Nos scènes*



3.2. Création de la scène principale

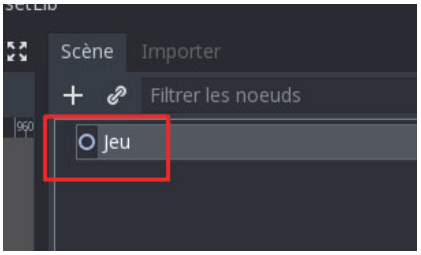
Nous allons maintenant créer la scène principale qui sera le `jeu`. Dans cette scène principale, nous instancierons nos deux scènes précédentes. Créez donc une nouvelle scène vide. Vous devriez avoir trois onglets :

Figure 3.7 : *Nos trois scènes*



Dans votre nouvelle scène vide, ajoutez un nœud avec Ctrl+A. Choisissez NODE2D qui est un nœud vide. Il faut toujours qu'un nœud principal englobe l'ensemble de vos autres nœuds. Renommez-le en `Jeu`.

Figure 3.8 : *Nœud principal*

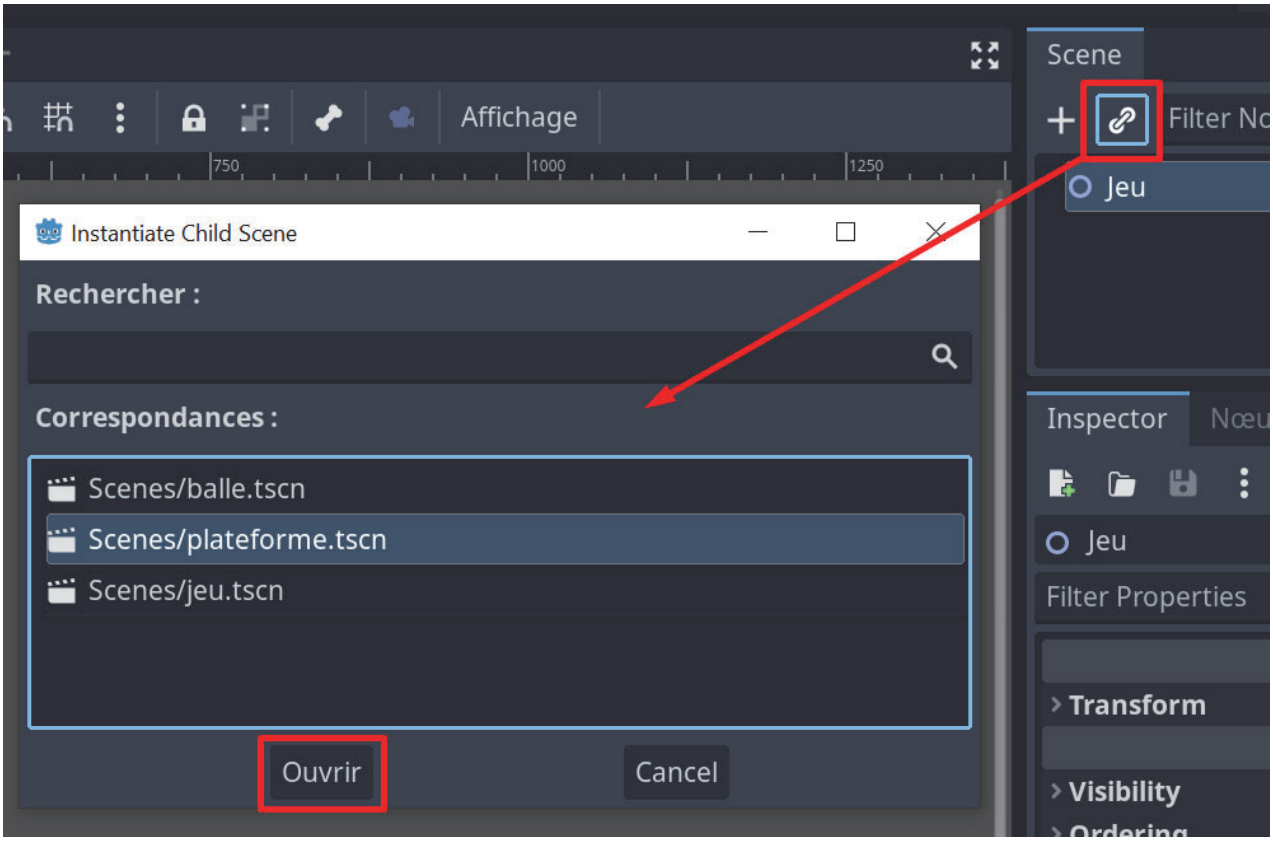


Sauvegardez votre scène dans le dossier des scènes. Nous allons maintenant pouvoir passer à l'instanciation.

3.3. Instanciation de scènes dans Godot

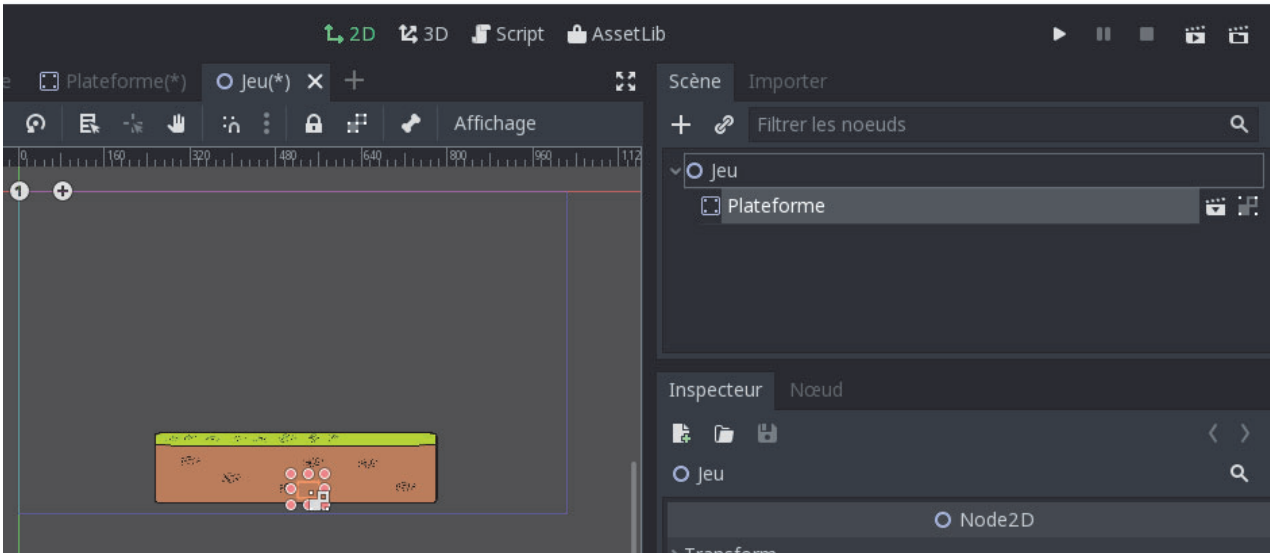
Pour instancier les deux scènes précédemment créées dans notre scène principale, cliquez sur le bouton d'instanciation en haut à gauche du panneau Scène. Dans la fenêtre qui s'ouvre, choisissez la scène que vous souhaitez instancier (dans notre cas, la plateforme).

Figure 3.9 : *Choix de la scène à instancier*



Votre plateforme apparaît alors dans votre scène. Positionnez-la en bas de l'écran.

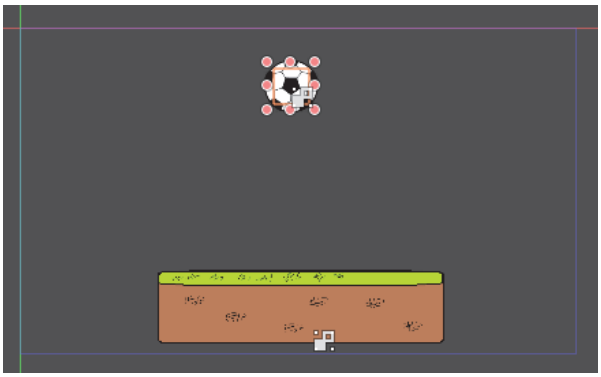
Figure 3.10 : *Instanciation de la plateforme*



Astuce › *Vous pouvez modifier la valeur SCALE de votre plateforme afin de réduire sa taille de moitié. Cela sera plus pratique pour bien visualiser la scène.*

Maintenant, instanciez la balle au-dessus de la plateforme :

Figure 3.11 : *Instanciation de la balle*



Attention › *Juste après avoir instancié la plateforme, elle sera sélectionnée. Si vous instanciez la balle, elle apparaîtra alors en tant qu'enfant de la plateforme. Ce n'est pas ce que nous voulons. Pensez à sélectionner le nœud de base afin que chaque objet soit enfant du jeu et non d'un autre objet instancié.*

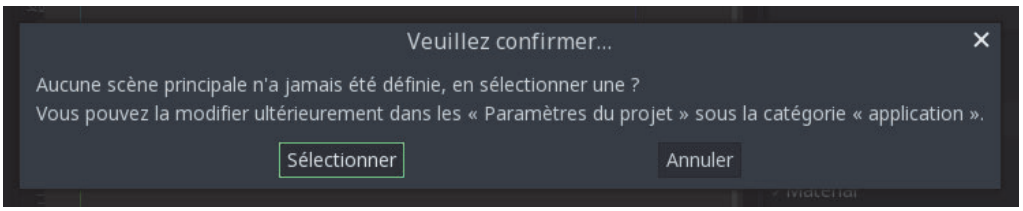
Comme vous pouvez le voir, j'essaie de centrer les éléments et de mettre la balle en hauteur pour tester le rebond. Le carré de couleur du viewport représente ce que verra le joueur dans la fenêtre de jeu. Essayez donc de positionner vos éléments dans cette zone visible par le joueur.

3.4. Lancement du jeu

Maintenant que notre projet est configuré, nous allons lancer le jeu. Un appui sur la touche F5 permet de lancer la scène principale.

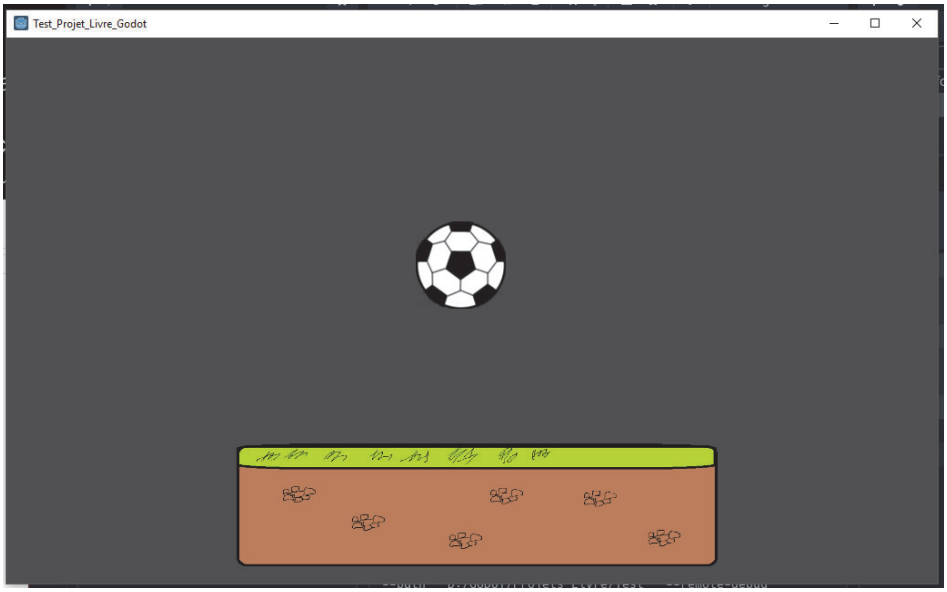
Actuellement, aucune scène principale n'est définie ; si vous appuyez sur F5, vous aurez le message suivant :

Figure 3.12 : *Sélection d'une scène principale*



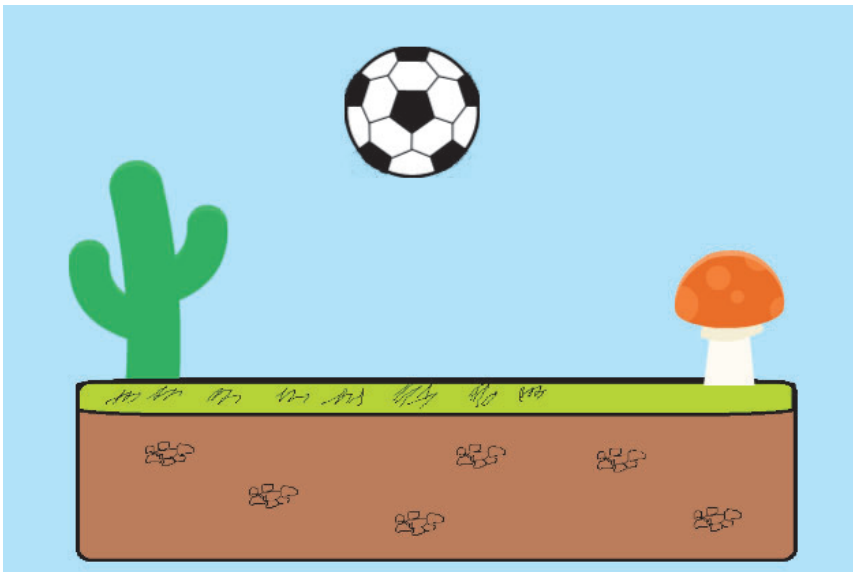
On nous demande de définir une scène principale qui doit être lancée avec F5. Cliquez sur le bouton SÉLECTIONNER, choisissez la scène JEU que vous venez de créer et validez. Une fois ceci fait, votre jeu se lancera et la balle rebondira sur le sol comme prévu.

Figure 3.13 : *Notre balle rebondissante*



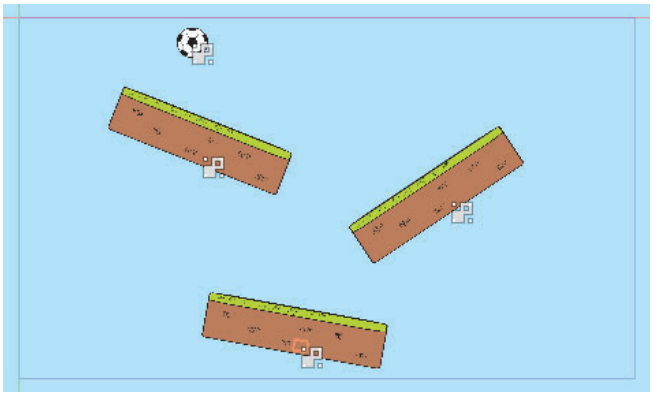
Vous avez donc vu comment créer des scènes et les incorporer à une scène principale pour former votre jeu. Afin de pratiquer de votre côté et vous familiariser avec Godot, je vous invite à ajouter des éléments décoratifs comme un ciel, des nuages et des plantes.

Figure 3.14 : *Ajout d'éléments décoratifs*



Vous pouvez également jouer avec la physique, modifier les paramètres du RigidBody, instancier plusieurs plateformes, modifier leur rotation pour voir comment réagit votre balle rebondissante et le moteur physique de Godot.

Figure 3.15 : *Jouons avec la physique !*



Pour le moment, nous ne pouvons rien faire d'autre que regarder ce qui se passe sur notre écran. Pour mettre en place des interactions et des événements, il allons avoir besoin d'écrire des scripts. Dans le chapitre suivant, nous allons voir comment se passe la création de scripts avec Godot.

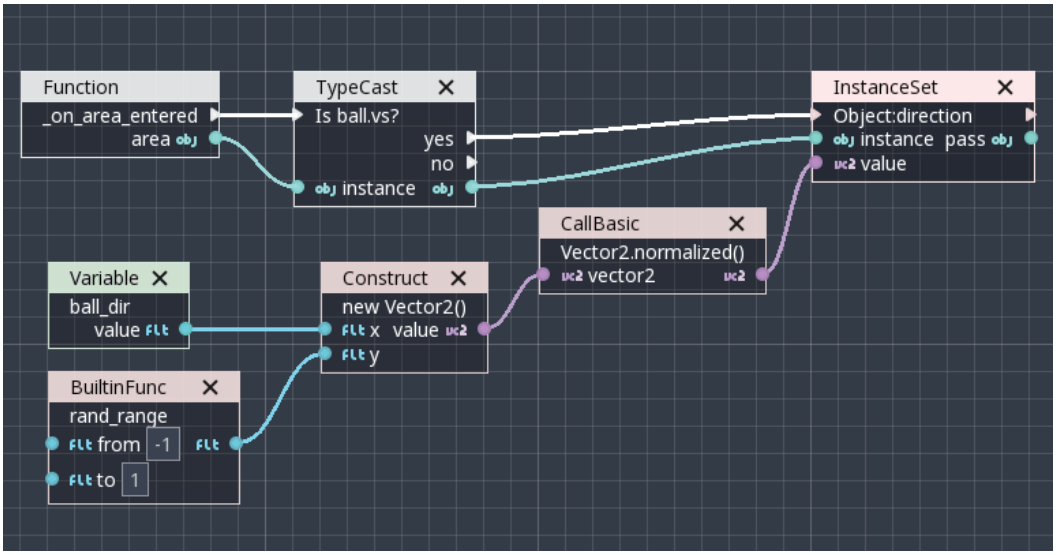
4

Initiation à la création de scripts avec Godot

Comme tous les moteurs de jeux, Godot permet la création de scripts. Les scripts permettent de mettre en place des interactions entre le joueur et le jeu. Nous pourrons récupérer les *inputs* (comme les entrées clavier/souris), gérer le comportement des objets (comme le déplacement du personnage ou des éléments à collecter) ou encore mettre en place les conditions de réussite/échec du jeu.

La création de scripts passe par de la programmation. Le langage de programmation historique de Godot est le GDScript, une sorte de Python. Ce langage est assez simple à prendre en main pour les débutants tout en offrant une grande puissance. À partir de la version 3 de Godot, d'autres langages de programmation ont été introduits, notamment le C# et un langage en *visual scripting* permettant de coder de façon visuelle par assemblage de blocs.

Figure 4.1 : *Le visual scripting sous Godot*



Depuis la version 4 de Godot, une nouvelle version de C#, plus performante, est proposée. C# a l'avantage d'être, dans la majorité des cas, plus puissant que GD Script.

En outre, c'est un langage très utilisé dans le monde du jeu vidéo (Unity, MonoGame, Stride, CryEngine, etc.). Il présente donc beaucoup d'atouts, et son emploi avec Godot progresse rapidement.

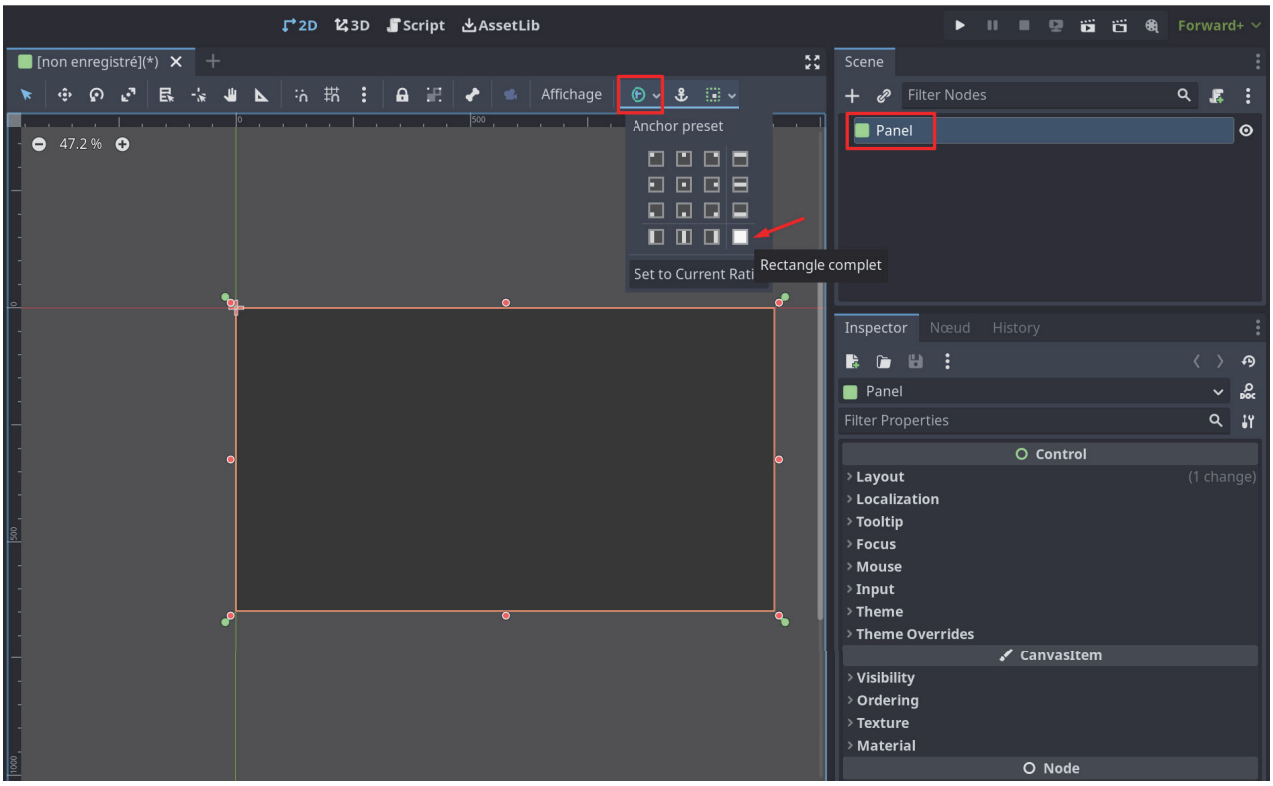
Bien que GDScript soit encore très populaire, nous n'utiliserons dans ce livre que le langage C#. Il s'agit du langage que je vous recommande car il va prendre beaucoup d'importance dans les années à venir. Il est aujourd'hui beaucoup plus documenté que par le passé et permet de faire autant que GDScript.

Dans ce chapitre, nous allons découvrir comment écrire nos scripts en C#.

4.1. Premier script

Commencez par créer une nouvelle scène vide afin de partir sur une base simple. Créez ensuite un nœud Panel (Ctrl+A) que vous agrandirez pour couvrir la taille de l'écran de jeu. Pour aller vite, vous pouvez utiliser les ancres prédéfinies.

Figure 4.2 : *Création d'un Panel*



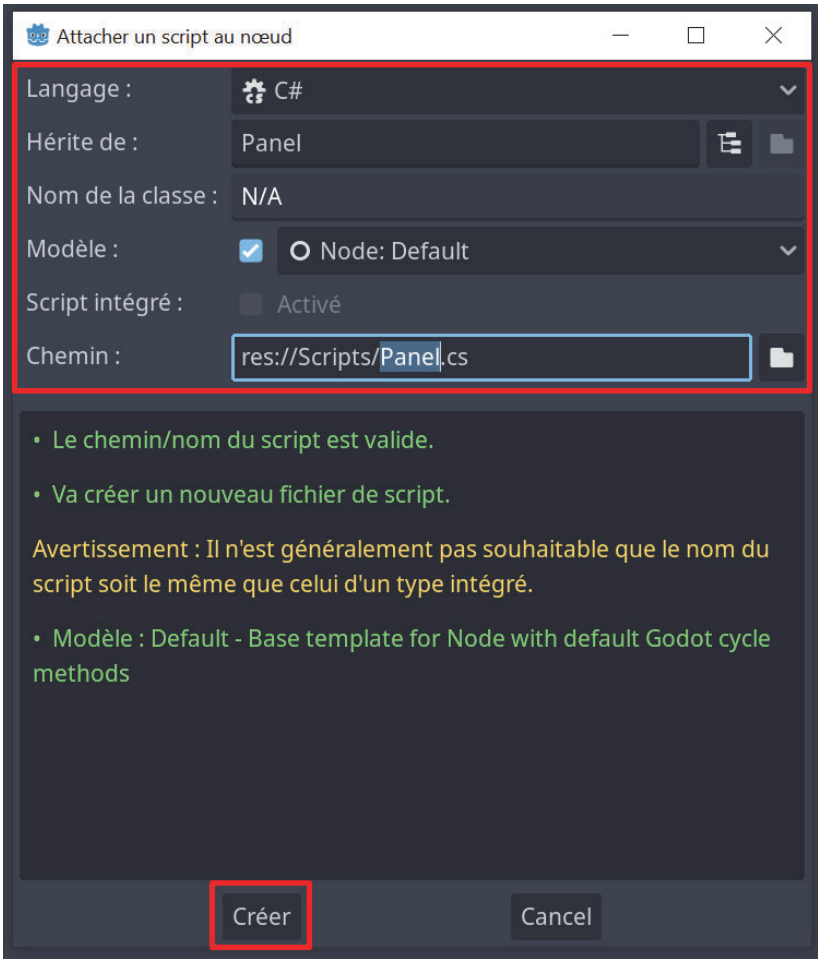
Note *› Le Panel est le composant qui est en général utilisé pour contenir des éléments d'interface comme du texte, des boutons et des champs textes. Un menu principal ou un menu pause sera par exemple conçu dans un panel.*

Nous avons créé un Panel car nous avons besoin d'un nœud principal pour pouvoir créer un script. Nous aurions pu créer n'importe quel autre type de nœud. Nous avons choisi un Panel car nous allons créer plus tard une UI (User Interface).

Lorsque vous avez un nœud dans votre scène, vous avez la possibilité de lui ajouter un script. Pour cela, avec le nœud sélectionné, cliquez sur l'icône Script se trouvant en haut à droite du panneau Scène.

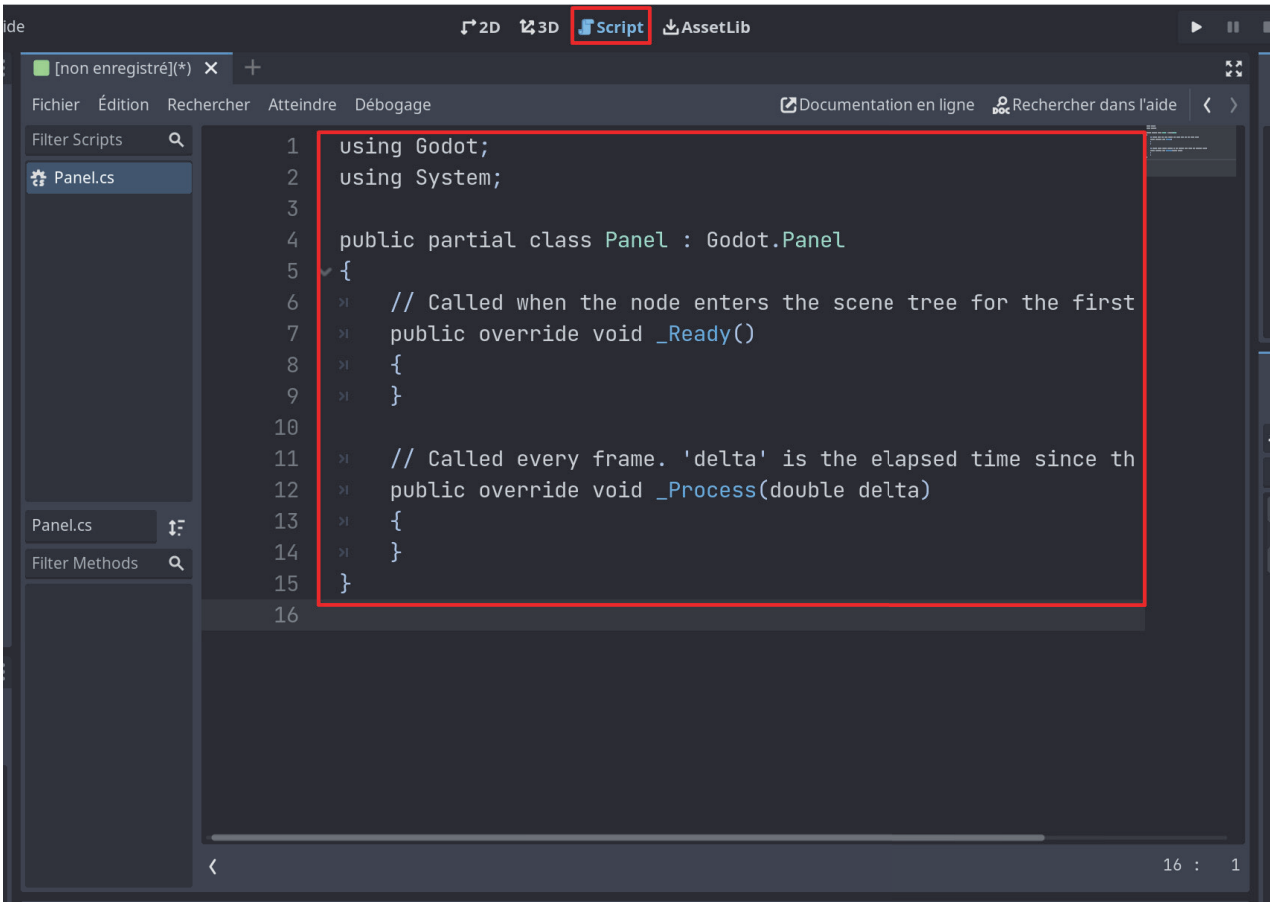
Lors de la création d'un script, vous devez sélectionner le langage ; nous allons choisir C#. Vous devez également préciser le modèle, gardez celui par défaut. Enfin, il vous faut spécifier un chemin : cliquez sur l'icône en forme de dossier afin de choisir le chemin d'enregistrement (je vous recommande de sauvegarder votre script dans un sous-dossier nommé `Scripts`). Une fois que tout est paramétré, cliquez sur CRÉER.

Figure 4.3 : *Configuration du script*



Lorsque vous validez, le script se crée et s'ouvre comme sur la Figure 4.4.

Figure 4.4 : *Script par défaut*



L'espace de travail SCRIPT s'est automatiquement ouvert. Quelques lignes de code sont déjà écrites par défaut. Voici quelques explications pour comprendre :

`using` est un mot clé qui permet d'inclure des fonctionnalités à notre script. Ici, nous incluons les fonctionnalités de Godot et du système.

`class` est le mot clé permettant de créer une classe (le code de notre objet) et `:Godot.Panel` d'indiquer que cette classe hérite des propriétés de `Panel` (car notre nœud est un Panel). L'héritage permet à l'enfant (le script) d'accéder aux propriétés du parent (ici Panel). Cela nous permettra de déclencher des événements propres au Panel si besoin.

Dans le code présent par défaut, vous remarquerez la présence de deux fonctions (`_Ready` et `_Process`).

FONCTION

Une fonction est une partie du programme qui exécute des opérations bien précises. Par exemple, nous pourrions créer une fonction `Soustraire` qui prendrait en paramètres `nombre1` et `nombre2` et qui retournerait la soustraction de ces deux nombres. Cette fonction pourrait ressembler à cela :

```
public int Soustraire(int nb1, int nb2)
{
  return nb1 - nb2;
}
```

Le mot clé `public` indique que la fonction est publique et qu'elle peut être appelée de n'importe où et par n'importe qui. L'inverse serait l'utilisation du mot clé `private` qui lui indique que la fonction ne peut être appelée que par la classe elle-même.

Le mot clé `return` permet de renvoyer une valeur. Une fonction peut être appelée par son nom, dans notre cas ce serait : `Soustraire(10,5);`. Le résultat serait alors 5.

`_Ready()` est une fonction particulière car elle s'exécute au lancement du programme. C'est la première fonction qui est lancée à l'ouverture de la scène.

`_Process(double delta)` est également une fonction très importante, car elle tourne en boucle. Cela signifie que si votre jeu tourne en 60 images par seconde, elle s'exécutera 60 fois par seconde. Son paramètre par défaut `delta` correspond au temps écoulé entre deux frames (entre deux images). Il permettra par exemple de calibrer vos animations pour que leur vitesse soit la même indépendamment de la puissance de l'ordinateur qui exécute le programme. Ce sera très utile par exemple pour les animations, afin de faire avancer le personnage de façon fluide.

Enfin, vous verrez dans les différents programmes, lors de la déclaration des fonctions, les mots clés `void` ou encore `override`. `void` signifie vide. Une fonction déclarée comme étant `void` ne retourne rien. Elle s'exécute mais ne retourne aucune valeur. Quand on parle de valeur de retour, il s'agit d'une valeur traitée par une fonction et renvoyée pour être utilisée ailleurs dans le code. Par exemple, une fonction `Additionner` devrait retourner une valeur qui serait le résultat de l'addition. `override`, quant à lui, est un mot clé qui permet de réécrire le comportement d'une fonction déjà existante. Dans le cas de `_Ready` par exemple il s'agit d'une fonction présente de base avec Godot. Pour la réécrire ou la redéclarer, on utilise `override`.

Voilà pour ce petit tour du code présent par défaut dans un script C#.

4.2. Indentation

Lorsqu'on écrit du code, il faut respecter certaines règles d'indentation. L'indentation consiste à ajouter des décalages (tabulations) en début de ligne. Par exemple, le code écrit dans un bloc (entouré d'accolades) doit être décalé d'un cran vers la droite. Le but étant de rendre le code plus facilement lisible. C'est pour cela que les fonctions `_Ready` et `_Process` sont décalées vers la droite dans le bloc principal.

4.3. Variables et constantes

Les *variables* sont utilisées pour stocker des valeurs qui peuvent changer au cours du temps contrairement aux *constantes* qui ne changent pas. Une variable peut stocker une valeur numérique, textuelle ou d'autres types de données.

Selon le type de donnée stockée, nous utiliserons un type de variable différent. Le type `int` permet par exemple de stocker un nombre entier. `float` permet de stocker un nombre à virgule, `bool` permet de stocker un booléen (vrai ou faux), `string` permet de stocker une chaîne de caractère.

Vous pouvez par exemple créer une variable `age` qui stocke l'âge de l'utilisateur. Pour créer une variable, nous utilisons le type `int` puis nous donnons un nom et une valeur à la variable :

```
int age = 20;
```

Note › *Vous remarquerez que la ligne se termine par un point virgule. En C# (comme dans de nombreux langages de programmation), chaque instruction se termine par un point virgule. L'oublier fera planter votre programme.*

Une constante est déclarée de la façon suivante. Par convention, le nom d'une constante est en majuscules.

```
const int MACONSTANTE = 10;
```

Pour vous entraîner, vous pouvez créer une fonction qui modifie le contenu d'une variable puis en retourne la valeur :

```
public int ChangerAge(int nouvelleValeur)
  {
    age = nouvelleValeur;
    return age;
```

```
}
```

Ce bout de code n'est pas vraiment utile tel quel mais c'est pour que vous compreniez qu'une fonction peut agir sur une variable et on peut imaginer des opérations plus complexes basées sur ce principe.

4.4. Commentaires

Les commentaires permettent d'écrire du texte dans votre code afin de décrire votre programme. C'est très précieux non seulement pour aider les autres programmeurs qui travailleront sur votre script mais aussi vous aider à vous souvenir du fonctionnement de votre programme si vous y revenez deux mois après. Pour écrire un commentaire, utilisez simplement un double slash `//` :

```
// Ceci est un commentaire
```

Les commentaires sont ignorés par Godot et ne sont donc utiles qu'aux programmeurs.

4.5. Conditions

Les conditions sont très utiles en programmation. Elles permettent de tester des égalités ou des différences afin d'exécuter une fonction ou une alternative. Par exemple, nous pouvons tester l'âge de l'utilisateur pour afficher une information ou une autre.

Nous utilisons le mot clé `if` pour écrire une condition. Nous faisons ensuite un test et si celle-ci est remplie, la fonction s'exécute ; sinon (`else`), une autre fonction peut être exécutée :

```
int age = 10;

  public override void _Ready()
  {
    if (age >= 18)
    {
      // L'utilisateur est majeur
    }
    else
    {
      // L'utilisateur est mineur
    }
  }
```

Il est possible de créer des conditions plus complexes mais nous y reviendrons par la suite.

4.6. Afficher du texte dans la console

La console permet d'afficher des valeurs pour vérifier le bon fonctionnement de votre code et déboguer votre programme. La fonction `GD.Print` permet d'afficher du texte dans la console. `GD.Print` s'utilise de la façon suivante :

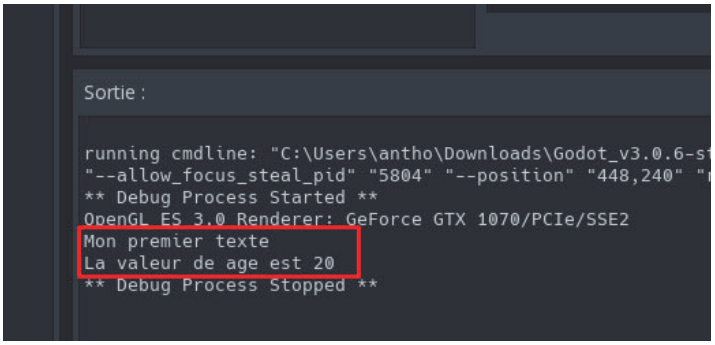
```
int age = 10;

  public override void _Ready()
  {
    GD.Print("Exemple de texte");
    GD.Print("La valeur de age est : " + age);
  }
```

Entre parenthèses, vous indiquez le texte à afficher.

À la seconde ligne, nous utilisons le `+` pour ajouter une information à notre texte, ici nous ajoutons la valeur de la variable `age`. Vous pouvez donc afficher du texte, la valeur d'une variable ou bien les deux en même temps.

Si vous sauvegardez votre script avec Ctrl+S (pensez à enregistrer votre script dans un dossier `Scripts` pour garder un projet organisé), vous pourrez l'exécuter en appuyant sur F6. Votre programme devrait se lancer et le résultat devrait apparaître en bas de l'écran dans la console de sortie.

Figure 4.5 : *Utilisation de la console de sortie*



Note › *F5 permet de lancer la scène définie comme étant la scène par défaut. F6 permet de lancer la scène actuellement ouverte pour la tester à la volée.*

Le texte apparaît dans la console : le script fonctionne conformément à nos attentes. N'hésitez pas à utiliser cette console lorsque vous programmez pour vérifier que les valeurs obtenues sont bien les valeurs attendues.

5

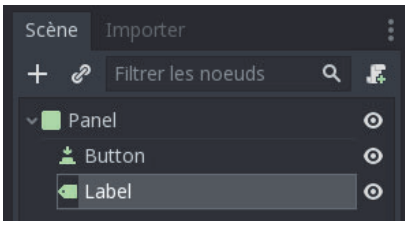
Plus loin avec C#

Dans le chapitre précédent, nous avons vu les bases de la programmation avec C#. Voyons maintenant comment connecter des éléments visuels entre eux et récupérer un événement utilisateur.

5.1. Création d'un bouton

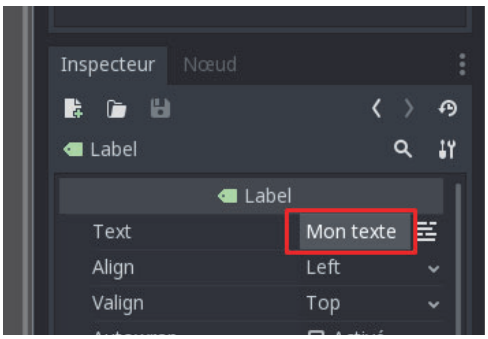
Nous allons continuer de travailler sur la scène précédente qui contient déjà un Panel et nous allons y rajouter deux nœuds : un Label qui affichera un texte et un bouton (Button).

Figure 5.1 : Ajout d'un Label et d'un bouton



Cliquez sur le Label et ajoutez un texte par défaut en passant par l'inspecteur et sa propriété TEXT.

Figure 5.2 : Texte du Label



De la même façon, ajoutez un texte par défaut sur votre bouton en utilisant l'inspecteur. Vous devriez avoir un résultat similaire à celui de la Figure 5.3.

Figure 5.3 : Notre interface utilisateur

Note *› Dans cet exemple j'ai centré les éléments dans le panel. Par défaut les éléments se créent dans le coin supérieur gauche. Nous devons les repositionner pour bien les voir et éviter qu'ils se superposent.*

Pour le moment, le texte est un peu petit mais cela suffira pour notre test ; nous verrons plus tard comment personnaliser la police d'écriture.

Nous allons programmer le bouton de sorte à ce que lorsque l'utilisateur cliquera dessus, le texte du Label soit modifié. Pour cela, retournez dans le script du Panel précédemment créé et supprimez tout le code de la classe. Il devrait alors vous rester les using et une classe vide :

```
using Godot;
using System;

public partial class Panel : Godot.Panel
{

}
```

Retournez sur la vue 2D, cliquez sur le bouton pour afficher l'inspecteur puis sur l'onglet NŒUD [Node] juste à droite de l'onglet INSPECTEUR [Inspector] :

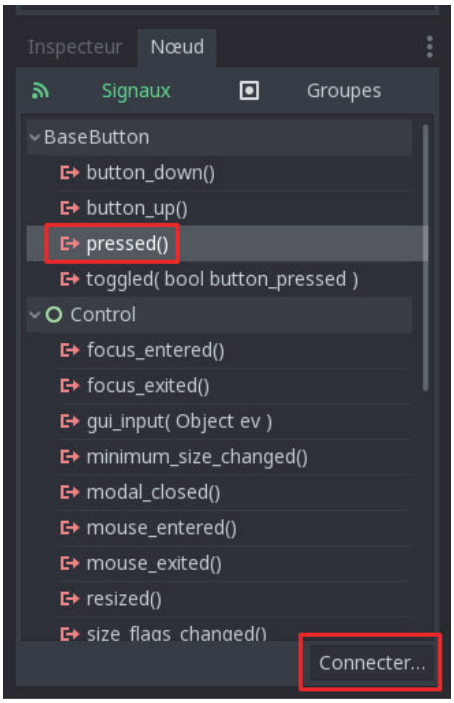
Figure 5.4 : Onglet Nœud

Si ce n'est pas le cas par défaut, cliquez sur SIGNAUX juste en dessous de l'onglet NŒUD. Ici vous retrouverez tous les signaux qui peuvent être associés à l'objet sélectionné, c'est-à-dire notre bouton.

Note *› Les* signaux *permettent aux nœuds d'envoyer des messages qui peuvent être écoutés par d'autres nœuds qui pourront eux-mêmes y répondre. Cela évite de vérifier continuellement si un bouton a été pressé : le signal est envoyé lorsque l'événement se produit, les autres nœuds le reçoivent et peuvent alors réaliser une action.*

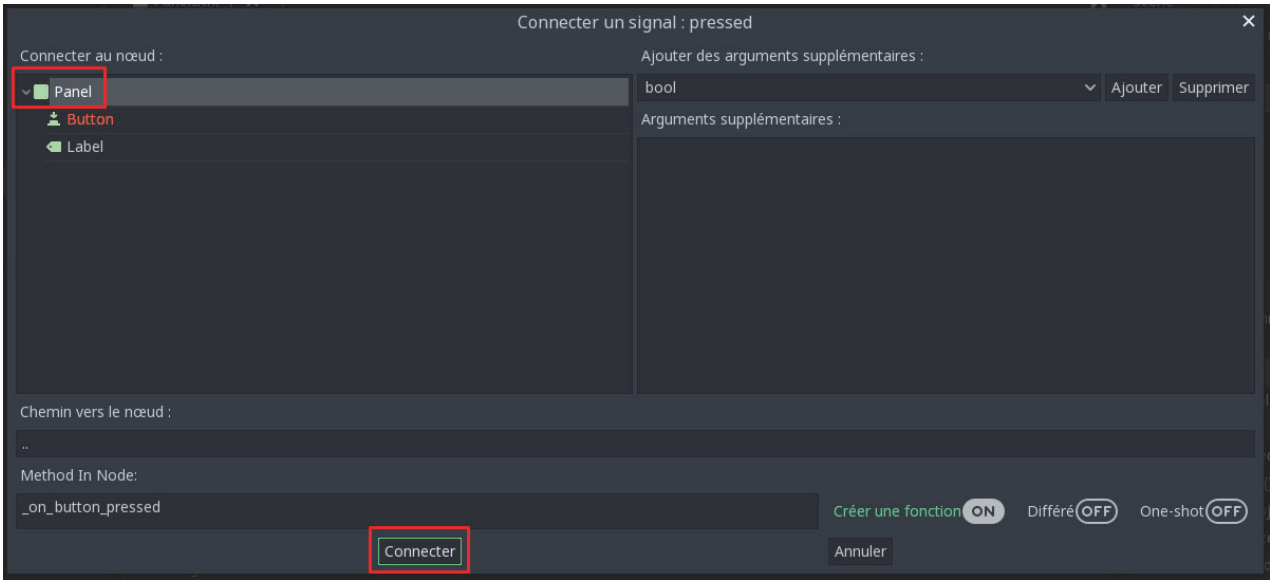
La Figure 5.5 montre la liste des signaux utilisables. Nous allons utiliser le signal `pressed()`, qui est déclenché lorsqu'un bouton est pressé. Sélectionnez ce signal et cliquez sur CONNECTER.

Figure 5.5 : *Connexion d'un signal*



Lorsque vous faites cela, Godot vous demande quel nœud doit être relié au signal. Sélectionnez le parent PANEL puis cliquez sur CONNECTER.

Figure 5.6 : *Connexion du signal à un nœud*



Une fois le signal connecté, le script réapparaît et une nouvelle fonction est créée. Il est possible que cette fonction soit créée en dehors de la classe (hors des accolades), si c'est le cas, déplacez le bout de code dans la classe.

```
private void _on_button_pressed()
{
  // Replace with function body.
}
```

Cette fonction sera déclenchée lorsque le bouton sera pressé. Le commentaire nous invite à écrire le code que nous souhaitons exécuter avec cette fonction. Dans notre cas, nous voulons accéder par script à la propriété texte du Label. Pour cela, nous devons dans un premier temps accéder au Label pour ensuite accéder au texte.

Notre script actuel se trouve sur le Panel qui est le nœud parent. Label (le nœud que nous souhaitons atteindre) est un enfant de Panel. Pour accéder à un nœud enfant, nous utilisons la fonction `GetNode` qui prend en paramètre le nom du nœud que nous souhaitons récupérer :

```
GetNode<Label>("Label");
```

Note › *La fonction `GetNode` permet de récupérer un nœud via son nom. Vous remarquerez que juste avant de spécifier le nom, nous devons ajouter `<Label>`. Cela permet d'indiquer quel est le type de l'élément ciblé. Grâce à cela, Godot sait que l'objet recherché `Label` est de type `Label`. Grâce à ce type, Godot sait ce qu'il est possible de faire avec l'élément. Ici, on sait par exemple qu'un label a une propriété `Text` qui est accessible.*

Une fois que nous sommes sur le bon nœud, nous pouvons utiliser la propriété `Text` pour modifier le texte.

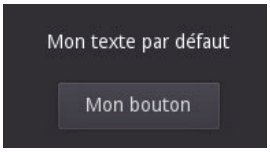
```
GetNode<Label>("Label").Text = "L'utilisateur a cliqué sur le bouton";
```

Le script complet de cet exemple est le suivant :

```
using Godot;
using System;

public partial class Panel : Godot.Panel
{
  private void _on_button_pressed()
  {
    GetNode<Label>("Label").Text = "Le texte modifié";
  }
}
```

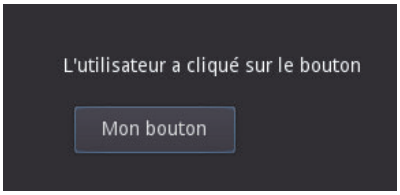
Testons maintenant notre programme. Sauvegardez le tout, appuyez sur F6 pour lancer la scène et regardez l'écran :

Figure 5.7 : *Avant le clic*



Ceci correspond à l'écran initial. Maintenant cliquez sur le bouton et observez le résultat obtenu :

Figure 5.8 : *Après le clic*

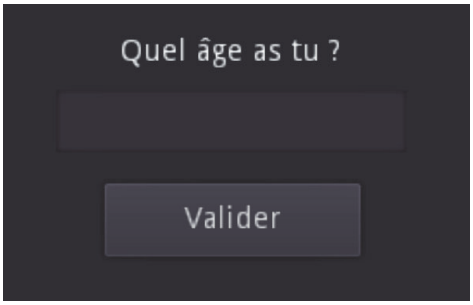


Nous avons bien réussi à modifier le texte au clic sur le bouton. Pour récapituler, nous avons créé les nœuds, lié le bouton au panel via un signal, accédé au Label depuis le Panel avec la fonction `GetNode` et modifié le texte de ce label.

5.2. Récupérer un texte saisi par l'utilisateur

Pour terminer, je vous propose de réaliser un petit exercice. Son but est de modifier notre interface afin d'y ajouter un composant TextEdit qui permet à l'utilisateur de saisir du texte. Vous demanderez à l'utilisateur quel est son âge et le résultat s'affichera au niveau du texte. Voici ce que vous devez créer :

Figure 5.9 : *Exercice à réaliser*

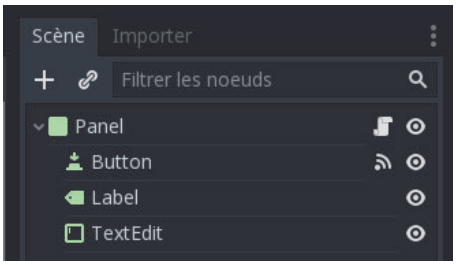


À vous de jouer !

Solution de l'exercice

Pour réaliser cet exercice, vous pouvez garder la scène précédente. Ajoutez-y simplement un TextEdit pour que l'utilisateur puisse saisir son âge.

Figure 5.10 : *L'interface utilisateur*

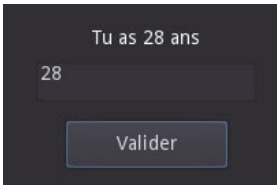


Modifiez ensuite le code afin de récupérer le texte rentré par l'utilisateur. Pour cela, utilisez de nouveau `GetNode` de la façon suivante :

```
private void _on_button_pressed()
  {
    GetNode<Label>("Label").Text = "Tu as "
      + GetNode<TextEdit>("TextEdit").Text + " ans.";
  }
```

Les + permettent de concaténer les mots pour créer une phrase composée de plusieurs valeurs. Testez votre programme pour vérifier son bon fonctionnement. Si tout fonctionne, vous devriez avoir le résultat ci-dessous.

Figure 5.11 : *Résultat de l'exercice*



Ces deux chapitres sur C# vous donnent un aperçu de la programmation sous Godot. Avec ces notions de base, vous êtes désormais en mesure de mieux comprendre son fonctionnement et vous lancer dans le développement de votre premier jeu.

Le premier jeu que nous allons créer sera un jeu 2D en vue de côté dans lequel le joueur incarnera un personnage capable de se déplacer, de sauter et de ramasser des objets.

6

Interview de Gilles Roudière, contributeur au projet Godot

Avant de passer à la création de jeux avec Godot Engine, voici une interview de Gilles Roudière, contributeur au projet Godot Engine, qui partage avec nous son expérience, sa vision des projets open-source et les forces de Godot Engine. Il vous donne des conseils pour participer au développement du moteur et vous former si vous avez envie de vous impliquer dans ce type de projets. Nous parlerons aussi de jeux vidéo.

Bonjour Gilles, peux-tu te présenter à nos lecteurs ?

Bonjour ! Je suis un des nombreux contributeurs de Godot. Ça fait environ six ans que je suis impliqué dans le projet et maintenant deux ans que je travaille à temps plein sur Godot et son écosystème. J'ai fait des études d'ingénieur en informatique et j'ai un doctorat en sécurité des systèmes d'information. Je suis originaire de la région Toulousaine, mais je vis maintenant perdu au centre de la France dans un village de 70 habitants (mais tout va bien, j'ai la fibre).

Quel est ton rôle au niveau du projet Godot Engine ?

J'ai surtout contribué à l'éditeur, et beaucoup à tout ce qui touche à la 2D. Ma plus grosse contribution reste la refonte du système de TileMap pour Godot 4.0. Au-delà de ça, j'aide aussi en faisant de la revue de code ou en aidant d'autres personnes à contribuer.

Qu'est-ce qui t'a poussé à t'impliquer dans ce projet ?

Même si je n'en ai pas fait mon métier directement, j'ai toujours aimé la conception de jeux vidéo. Quand j'avais dix ans, un ami et moi avions créé la démo d'un petit jeu fait sous RPG maker, c'était pas grand chose, mais c'était notre œuvre. En 2017, je me suis demandé si je pouvais créer un autre petit jeu (type *Metroidvania*) inspiré du même univers. J'ai alors cherché quel moteur de jeu utiliser. J'ai essayé Unreal, mais la compilation plantait. Puis j'ai essayé Unity, mais ça m'obligeait à coder sous Windows alors que je préfère Linux, et sa gestion de la 2D n'était pas terrible. Alors tout bêtement, j'ai un jour fait une recherche sur GitHub pour voir s'il n'existait pas une alternative open-source. C'est là que je suis tombé sur Godot.

Ça a été un peu un coup de cœur, j'ai pu faire un petit prototype en à peine quelques dizaines de minutes. L'éditeur était simple et intuitif et correspondait à ce que je cherchais.

Ceci dit, l'interface avait besoin de quelques modifications pour être un peu plus efficace. Par curiosité, j'ai alors jeté un coup d'œil au code, puis fait une *pull request* (une proposition de modification du code) avec quelques changements. Elle a été acceptée très vite, ce qui m'a poussé à continuer à contribuer. Depuis, je n'ai jamais arrêté.

Quel est l'intérêt de s'impliquer dans le développement d'un projet open-source ?

Je dirais que, pour un amateur, c'est essentiellement l'envie de contribuer à un bien commun et de rendre certaines technologies plus accessibles. Je trouve qu'il y a beaucoup de sens à travailler sur quelque chose qui bénéficie à tout le monde. Quant aux entreprises, certaines mettent des millions d'euros dans des solutions internes coûteuses, qui pourtant ne leur rapportent qu'assez peu directement. Pour réduire ces coûts, l'open source est une bonne solution, puisqu'il permet de mettre en commun le coût de développement et de maintenance des logiciels.

Quels sont les avantages de Godot par rapport à ses concurrents ?

On ne parle pas vraiment de *concurrence* pour Godot, puisque le moteur appartient en quelque sorte à tout le monde, nous ne sommes pas vraiment là pour la compétition. Ceci étant dit, par rapport à d'autres solutions, je dirais que son aspect open-source, et notamment sa licence très permissive (MIT) restent son principal atout. Ensuite, son éditeur intégré, son moteur 2D dédié, ainsi que son langage de script simple qui permet des itérations rapides. En général, Godot est un moteur de jeu qui vise en premier lieu la simplicité d'utilisation, tout en laissant la porte ouverte aux améliorations de performances là où elles seraient nécessaires.

Que penses-tu de la nouvelle version de Godot ?

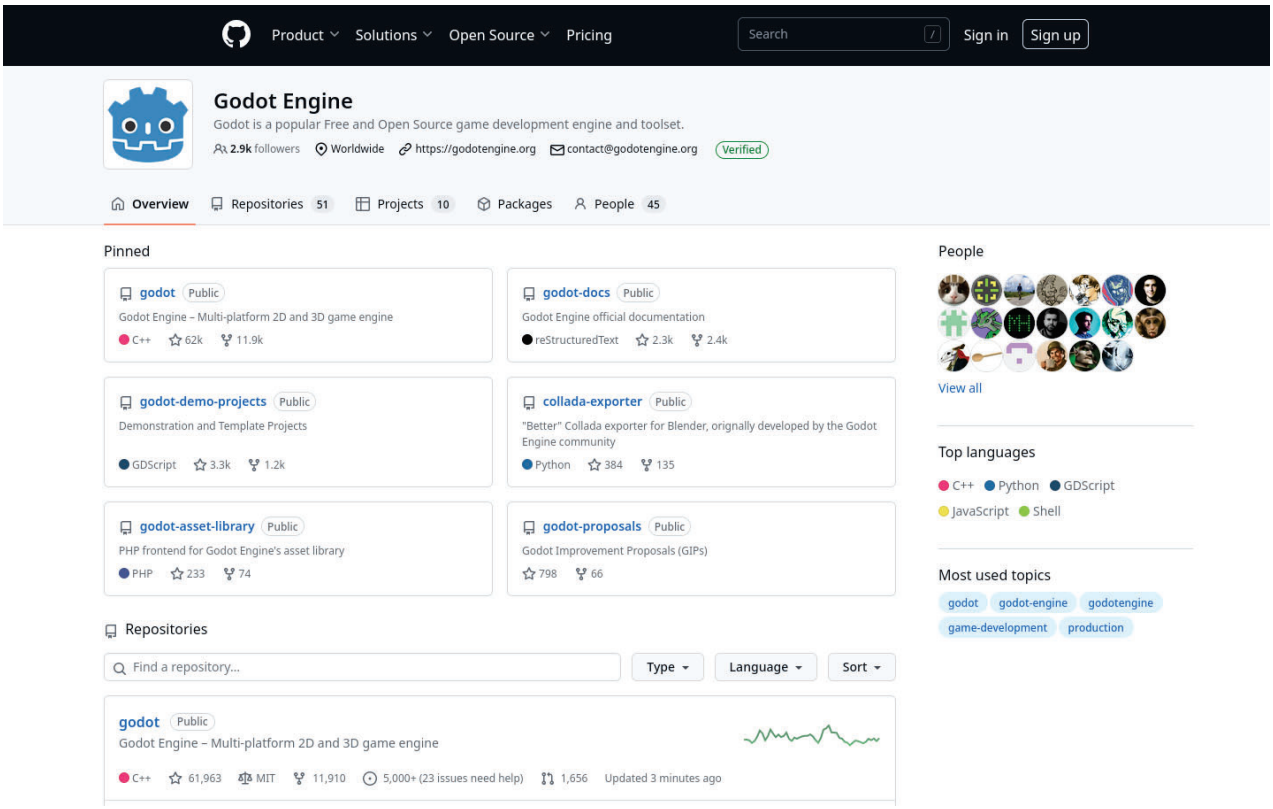
Elle est incroyable ! De nombreuses améliorations ont été faites. La version4.0 est probablement un peu moins performante et stable que 3.x, mais une énorme refonte du code a été faite sur des systèmes internes (moteur de rendu, moteur physique, etc.). Elle propose aussi plein de nouvelles fonctionnalités. Cette nouvelle version est une base solide pour permettre dans les prochaines versions mineures (4.1, 4.2, etc.) des boosts de performances significatifs.

As-tu des conseils pour ceux qui souhaitent s'investir dans le développement du moteur ?

Godot est devenu un très gros projet, alors les demandes de nouvelles fonctionnalités sont acceptées au compte-goutte. N'hésitez pas à ouvrir une proposition sur le projet

GitHub dédié et à venir discuter avec les autres contributeurs sur notre plateforme de chat pour savoir si votre nouvelle fonctionnalité a des chances d'être acceptée. Sinon, commencez par la résolution de bogues.

Figure 6.1 : *Dépôt GitHub du projet Godot Engine*



Parlons un peu jeux vidéo.

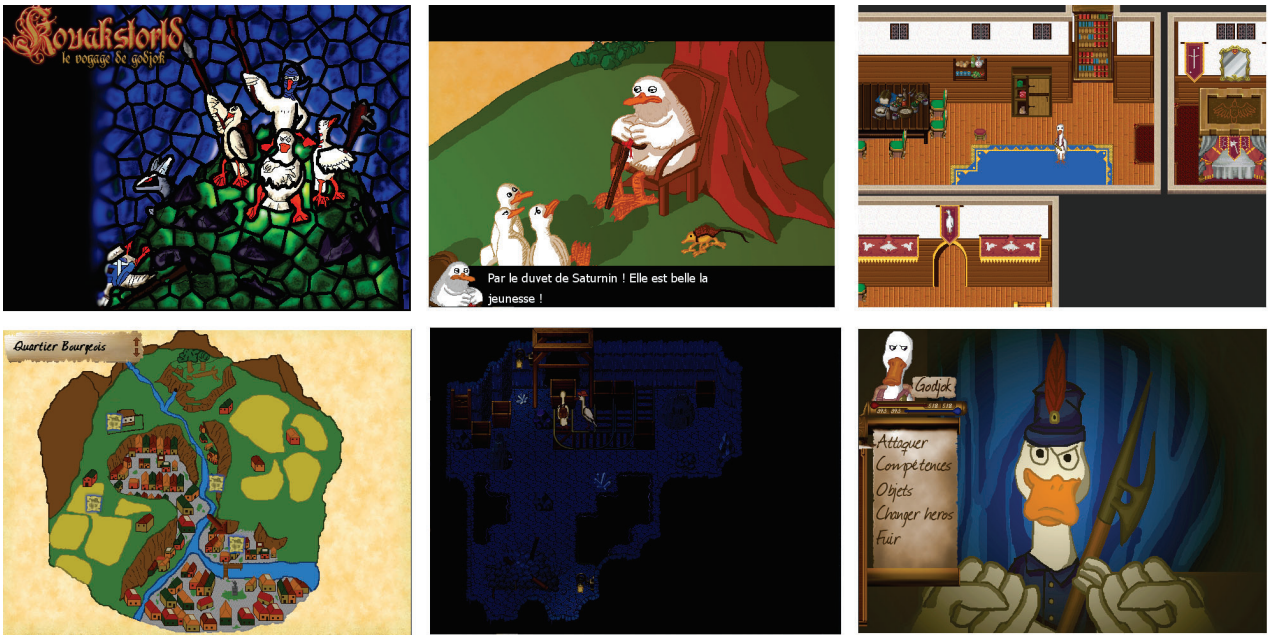
En dehors de ta participation au développement du moteur, as-tu eu l'occasion de travailler sur le développement de jeux ?

À part mes projets amateurs quand j'étais jeune, j'ai travaillé un temps sur un jeu en voxel art.

Aurais-tu des images, astuces ou anecdotes à partager ?

Oui, j'ai capturé quelques images de notre jeu réalisé quand j'étais enfant avec mes amis Romain et Quentin. Il s'appelait *Le voyage de Godjok* et était réalisé sur RPG Maker XP. C'est un jeu d'aventure qui se déroule dans le monde de Kouakstorld, peuplé de palmipèdes appelés Cou-tordus.

Figure 6.2 : *Un jeu d'enfant ;-)*



Quels sont tes jeux favoris ?

En solo, j'aime beaucoup les jeux d'aventures (*The Witcher 3*, *Zelda BOTW*) et les jeux d'énigmes (*Return of the Obra Dinn*, *The Witness*, *FEZ*). J'ai fait pas mal de classiques aussi (*Hades*, *Doom*, *Shovel Knight*, *Portal*). Je joue un peu à tout tant que ce n'est pas trop exigeant ni trop d'investissement. En ce moment, je joue beaucoup en ligne avec des copains (Apex, League of Legend, Valheim, Factorio, Divinity Original Sin).

Quel est le jeu développé avec Godot qui t'a le plus impressionné ?

J'ai beaucoup aimé *The Case of the Golden Idol*, c'est un jeu d'énigmes fantastique.

Connais-tu des studios de développement de jeux qui utilisent Godot ?

Ah oui, il y en a beaucoup maintenant ! Surtout des petits studios indépendants. J'invite tout le monde à visiter la page SHOWCASE sur godotengine.org, il y a plein de pépites sorties ou à venir.

Connais-tu un site web répertoriant des informations, ressources ou éléments utiles pouvant aider les nouveaux développeurs qui souhaitent monter en compétence sur Godot ?

La documentation de Godot est probablement la ressource numéro un pour apprendre à utiliser Godot. Pour des cours plus complets et avancés, GDQuest fait un travail formidable et contribue beaucoup au projet. Sur YouTube, Heartbeast a aussi fait une série de tutoriels assez appréciée. Ceci étant dit, l'essentiel est trouvable sur godotengine.org.

Figure 6.3 : *The Case of the Golden Idol (par le studio letton Color Gray Games)*

Un petit mot pour la fin ?

Merci de m'avoir invité sur ces pages. J'espère que ce court témoignage, et bien sûr cet ouvrage, participeront à convaincre plus de monde que Godot est une alternative crédible aux moteurs de jeux propriétaires. Aussi, Godot a besoin de contributeurs dans plein de domaines, n'hésitez pas à venir nous rejoindre !

Développement d’un jeu 2D

Dans cette partie, nous développerons notre premier jeu avec Godot. Pour commencer en douceur, nous nous attellerons à la création d’un jeu en deux dimensions. En effet, les jeux 2D sont un peu plus simples à développer que les jeux 3D. Il s’agira d’un jeu en vue de côté dans lequel nous contrôlerons un personnage qui peut se déplacer, sauter et ramasser des objets. Le but est de voir comment se passe la création d’un jeu sous Godot, du début à la compilation finale.

7

Mise en place du projet

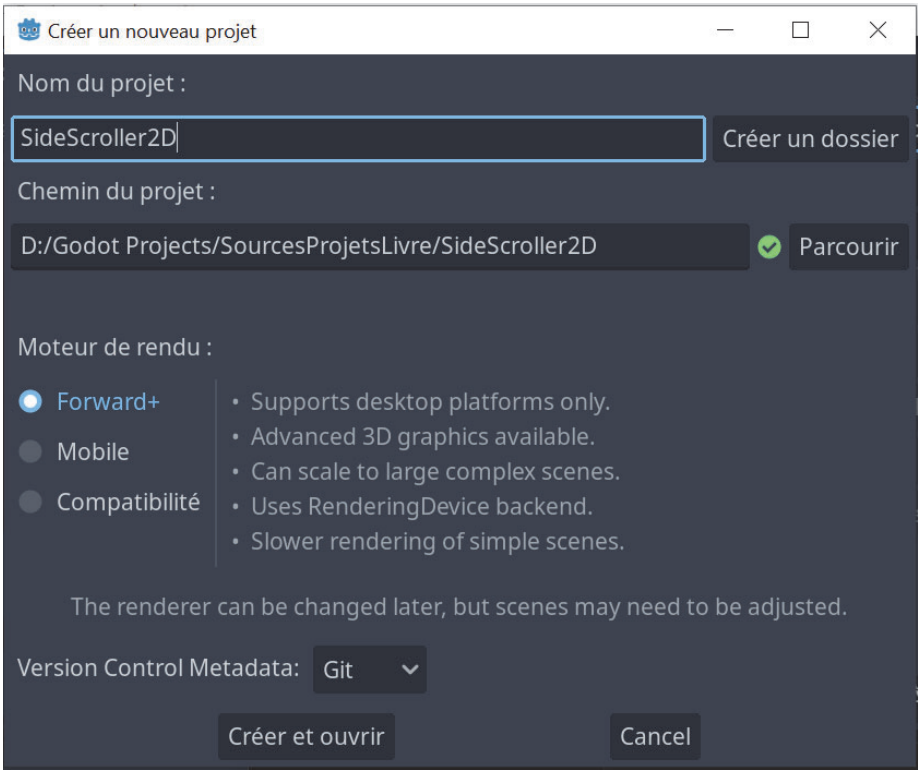
Nous allons donc développer notre premier jeu 2D. Dans ce jeu, nous aurons un personnage, des plateformes, des objets et des pièges. Le but sera de rejoindre la fin du niveau pour accéder au niveau suivant.

Dans ce chapitre, nous allons mettre en place le projet, c'est-à-dire créer le projet, récupérer des ressources, les importer et paramétrer tout cela.

7.1. Création du projet

La première chose à faire est de créer un nouveau projet. Lancez Godot et dans la fenêtre de démarrage, choisissez NOUVEAU PROJET. Donnez un nom à votre projet, choisissez un emplacement où le sauvegarder et validez en cliquant sur CRÉER ET OUVRIR.

Figure 7.1 : Création du projet



Dans mon cas, le projet sera stocké sur mon disque D et il portera pour nom `SideScroller2D` car il s'agit d'un Side Scroller.

Note *› Pour réaliser ce projet, vous utiliserez les ressources fournies avec ce livre. Vous aurez également accès au projet final. Essayez de tout faire par vous-même, ne regardez pas le projet final avant d'avoir vous-même créé votre jeu. Les sources sont là pour vous aider si vraiment vous avez un blocage. Pour ouvrir un projet existant avec Godot, cliquez sur le bouton IMPORTER au démarrage du logiciel et recherchez le fichier projet à importer.*

7.2. Trouver des ressources pour son jeu

Pour pouvoir créer un jeu, il nous faut des ressources afin d'avoir de la matière avec laquelle travailler. Ce que j'appelle des ressources, ce sont les assets que nous avons à disposition pour concevoir notre jeu, à savoir les textures, les sons, les modèles 3D, etc.

Avant de rassembler ces ressources, vous devez décider quel sera le style graphique du jeu. Voulez-vous faire des niveaux en pixel art (c'est-à-dire avec des éléments très pixellisés) ou en 2D bien lissés ? Ensuite, il vous faudra choisir les couleurs : est-ce que votre jeu aura une ambiance sombre ou au contraire des décors très colorés ?

Une fois que vous savez ce que vous souhaitez obtenir visuellement, vous devez vous procurer des ressources. Pour cela, plusieurs solutions existent. Soit vous êtes un artiste et vous créez vous-même vos ressources (vous dessinez les personnages, les décors...), soit vous avez des connaissances qui peuvent faire cela pour vous, soit vous achetez des ressources à un artiste. Nous allons partir du principe que dessiner n'est pas votre point fort et que vous ne maîtrisez pas les logiciels de dessin. Nous allons également partir du principe que vous n'avez pas dans votre entourage d'artiste et que vous ne souhaitez pas dépenser un centime pour réaliser votre jeu. Quelle solution nous reste-t-il ?

Comme je vous le disais en introduction de ce livre, Godot est un logiciel gratuit, libre et open-source. Cela signifie que le logiciel est mis à votre disposition gratuitement et que vous pouvez en faire ce que vous souhaitez. Comme pour Godot ou les autres logiciels libres, il existe des artistes très talentueux qui partagent gratuitement et librement de très nombreuses ressources graphiques. Des textures et décors 2D peuvent aussi être libres et gratuits. En général, ces ressources sont distribuées sous licence Creative Commons Zero ; cela signifie que vous êtes libre de les utiliser comme bon vous semble, même pour des projets commerciaux, sans rien demander au créateur de ces ressources.

De nombreux sites web proposent des ressources libres. OpenGameArt, que nous avons déjà vu en première partie, est un exemple de moteur de recherche de ressources libres. Je vais vous présenter deux autres sites d'artistes qui partagent ce qui se fait de mieux en matière de ressources 2D. Ces artistes sont très connus par les développeurs indépendants.

KENNEY

Kenney est un artiste très connu des indépendants. Il propose des milliers d'éléments à télécharger. Sa spécialité est la création de contenus 2D, mais il propose également des éléments 3D, des sons et musiques ou encore des éléments d'interface comme des boutons et menus. Ses ressources sont disponibles sur de nombreuses plateformes comme `itch.io`, ainsi que sur son site web.

Kenney propose tous ses assets gratuitement. Il est possible de les payer si vous voulez le soutenir. Tous ses assets sont dans le domaine public, cela signifie que vous pouvez les utiliser sans aucune restriction.

Voici un exemple de package proposé par Kenney :

Figure 7.2 : *Exemple de package créé par Kenney*

Pour notre exemple, nous utiliserons un package de Kenney. J'ai choisi l'`Abstract Platformer` (voir Figure 7.3). Vous pouvez bien entendu choisir un autre package, mais ce sera plus facile de suivre le livre en prenant le même.

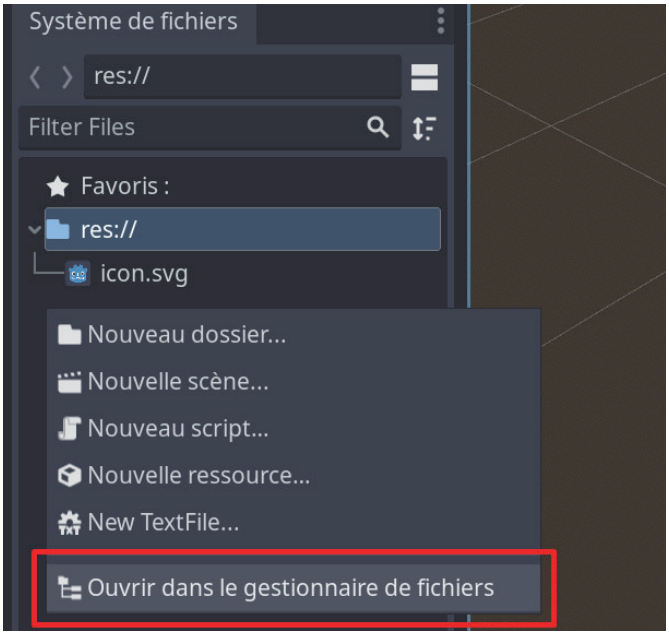
Figure 7.3 : *Abstract platformer*

Ce package sera parfait pour découvrir la création de jeux 2D avec Godot. Comme ces ressources sont libres, je vous les propose en téléchargement avec ce livre. Vous les retrouverez donc dans les fichiers sources.

7.3. Importer les ressources

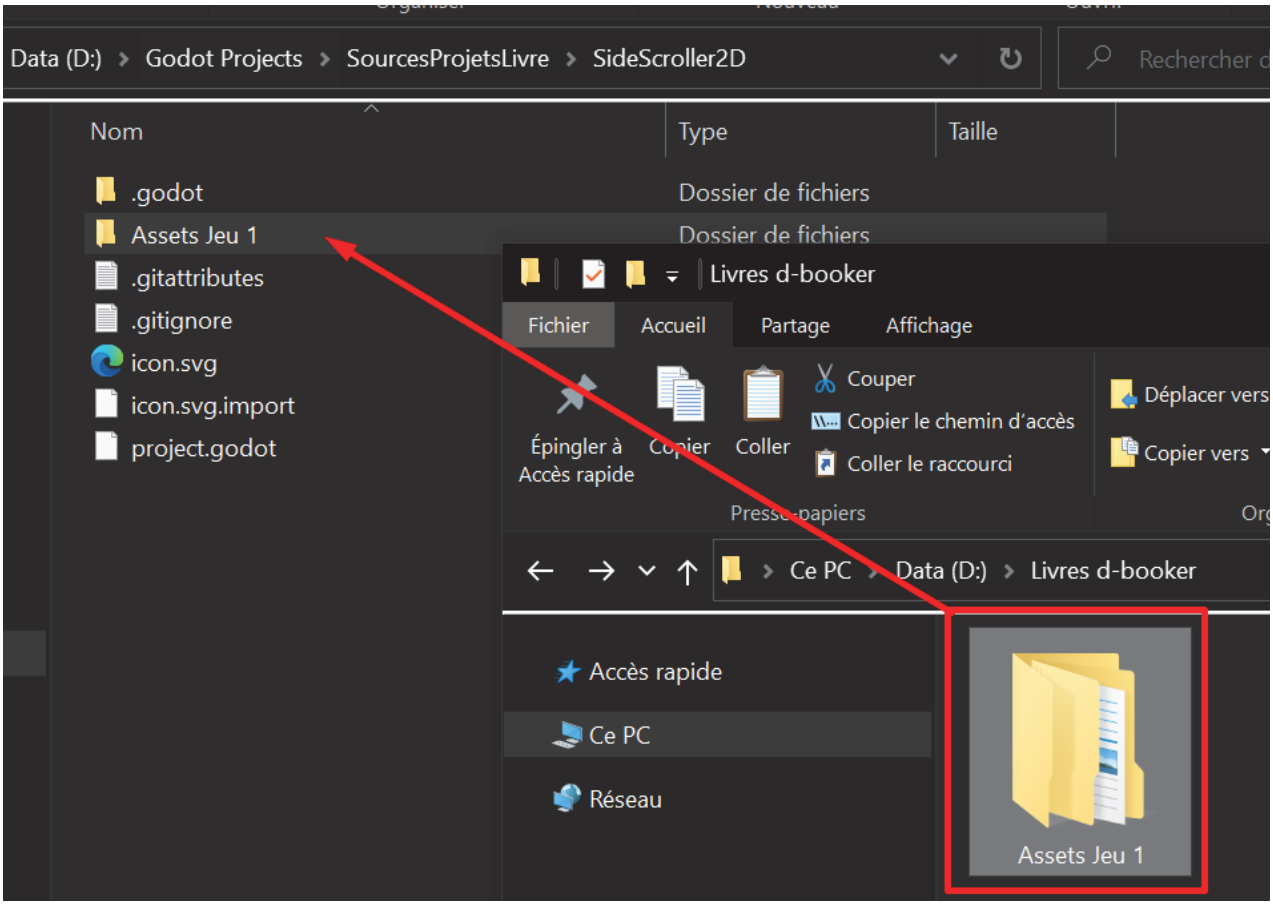
Vous avez créé un nouveau projet sous Godot et disposez de ressources (celles du livre ou d'autres acquises par vous-même). Pour les importer, cliquez droit dans le panneau SYSTÈME DE FICHIERS et sélectionnez OUVRIR DANS LE GESTIONNAIRE DE FICHIERS.

Figure 7.4 : *Affichage des fichiers du projet*



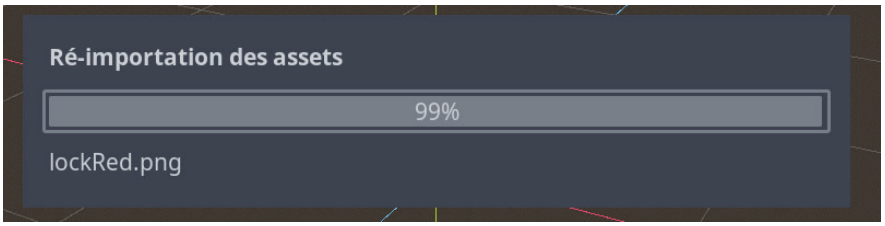
Cette action ouvre le dossier de travail de votre projet Godot. Copiez/collez ici le dossier contenant les assets que vous voulez utiliser pour créer votre jeu.

Figure 7.5 : *Copie des ressources dans le projet*



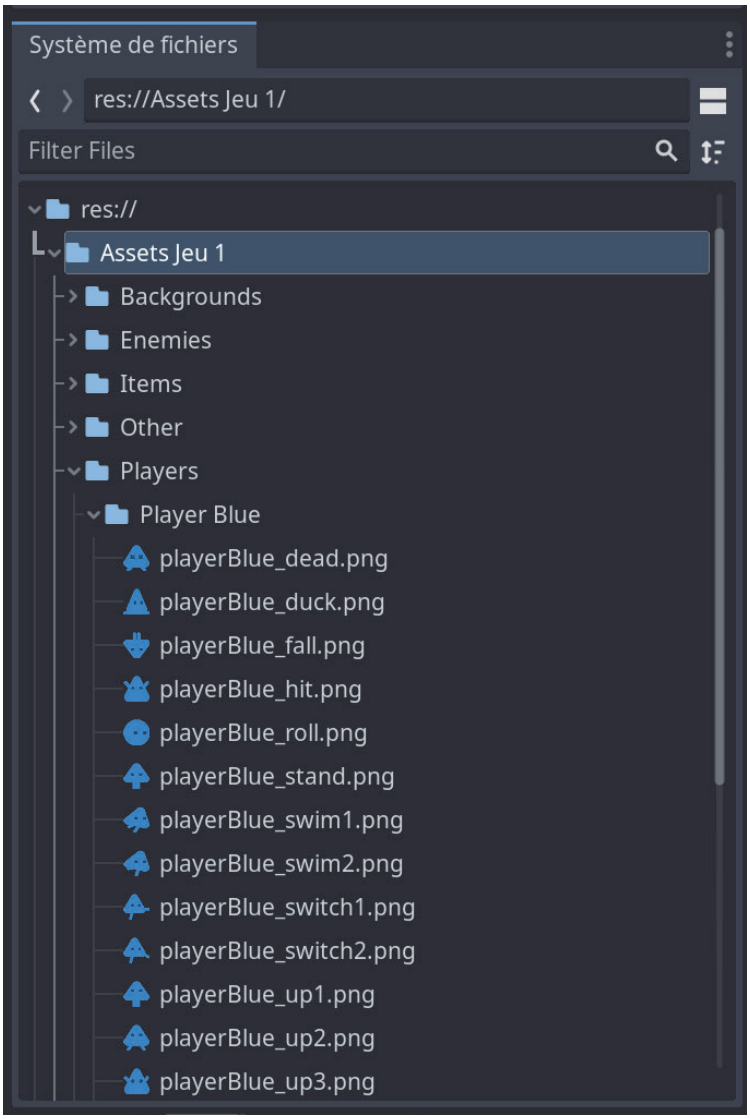
Lorsque vous retournez sur Godot, l'import se fait :

Figure 7.6 : *Import des ressources*



Cela peut demander du temps en fonction du nombre de ressources à importer. Une fois l'import terminé, l'ensemble des ressources seront disponibles dans votre projet Godot :

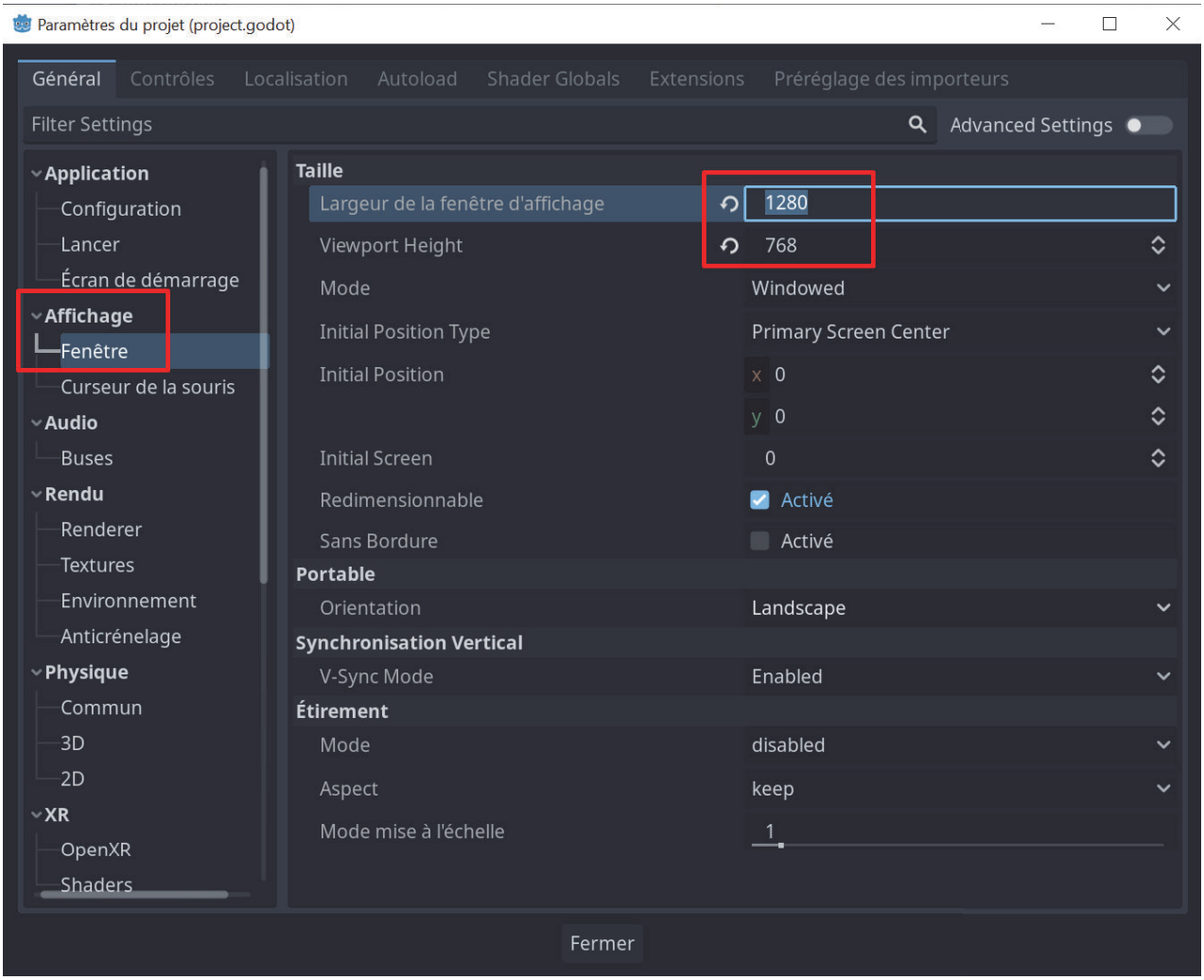
Figure 7.7 : *Ressources utilisables*



7.4. Paramètres du projet

Nous allons maintenant ajuster les paramètres de notre projet. Pour accéder aux paramètres, utilisez le menu PROJET/PARAMÈTRES DU PROJET. Dans le panneau de gauche, allez dans la rubrique AFFICHAGE/FENÊTRE. Vous pouvez y ajuster la résolution de votre jeu. Nous la réglerons sur 1280 × 768.

Figure 7.8 : *Résolution du jeu*

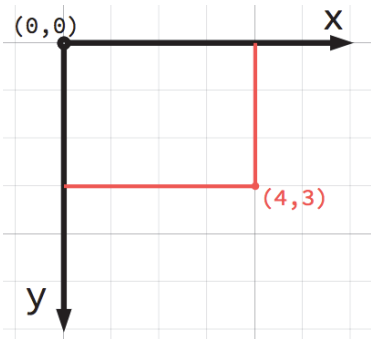


Je choisis cette résolution car nous allons utiliser (si vous utilisez le même package que moi, ce que je vous conseille) des tuiles de 64 pixels par 64. Avec cette résolution, nous aurons une taille idéale pour que le joueur puisse voir ce qu'il faut. De plus 1280 et 768 sont des multiples de 64 ce qui fait que nous aurons un écran de 20 cases par 12 cases très précisément. Cliquez ensuite sur FERMER pour terminer les modifications.

7.5. Système de coordonnées 2D

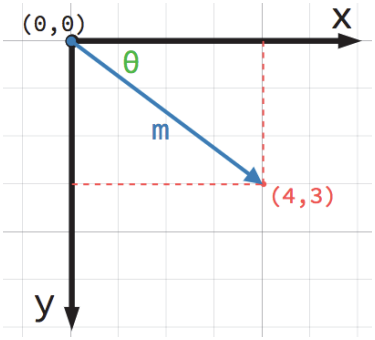
Avant de conclure ce chapitre, nous devons aborder les vecteurs et le système de coordonnées. En mathématiques et dans la plupart des moteurs de jeu, le système de coordonnées est celui que nous connaissons avec l'origine en bas à gauche. L'axe des x est horizontal et l'axe des y vertical. Lorsqu'on se déplace de $2x$, on se déplace de deux crans vers la droite et quand on se déplace de $3y$, on monte de trois crans. Avec Godot, cela est légèrement différent : l'axe des y est inversé. Lorsqu'on ajoute $1y$, on descend d'un cran. L'origine des coordonnées est en haut à gauche de l'écran. La Figure 7.9 sera peut-être plus parlante. Nous y avons placé un point en $4x$, $3y$.

Figure 7.9 : *Système de coordonnées dans Godot*



En 2D, chaque position ou chaque déplacement correspond à un vecteur 2D. Ce vecteur 2D (Vector2D) prend en paramètres deux nombres (x et y).

Figure 7.10 : *Vecteur 2D*



Cela peut sembler un peu flou mais nous aurons l'occasion d'y revenir. Pour l'instant, retenez simplement que l'origine est en haut à gauche de l'écran.

Nous voilà prêts pour commencer le développement de notre jeu. Au chapitre suivant, nous allons créer le personnage qui sera contrôlé par le joueur.

8

Création du personnage joueur

Le premier élément que nous allons créer sera le personnage qui sera incarné par le joueur. Cet élément est sans doute le plus important car ce personnage est toujours visible à l'écran. Il est au cœur du jeu, c'est lui qui vivra l'aventure de bout en bout. Nous devons donc y apporter le plus grand soin.

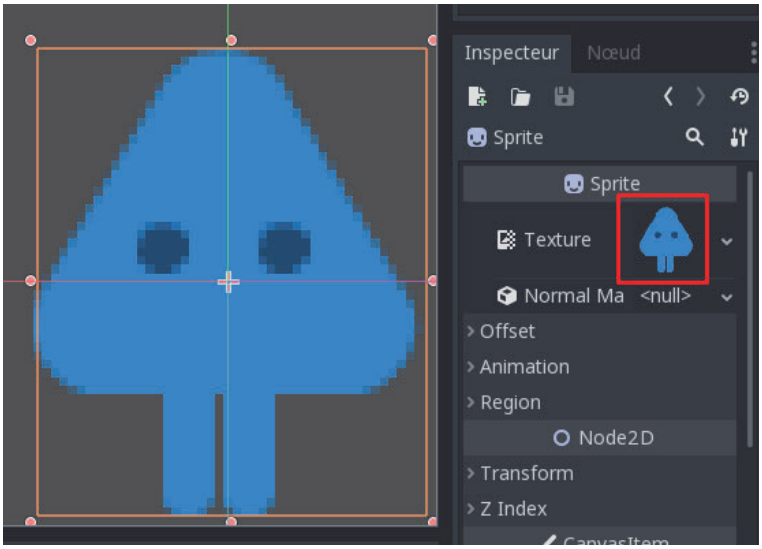
8.1. Création du nœud joueur

Nous allons nous pencher sur la création du nœud joueur. Notre personnage doit être capable de se déplacer et de gérer les collisions. Comme nous l'avons vu dans la première partie de ce livre, le composant physique adapté aux personnages est le CharacterBody2D. C'est donc celui-là que nous utiliserons. Pour la gestion des collisions, un CollisionShape2D fera parfaitement l'affaire. Enfin, notre personnage doit être visible à l'écran : il nous faudra donc un composant Sprite2D. Ce sont ces trois éléments que nous devons mettre en place a minima pour avoir un personnage fonctionnel.

Dans votre projet, créez une nouvelle scène puis créez un nouveau nœud. Choisissez le CharacterBody2D. Renommez-le en `Player` afin de commencer sur de bonnes bases. Ajoutez un Sprite2D en tant qu'enfant de ce nœud. Ce Sprite peut prendre en paramètre une texture. Ajoutez donc une texture qui représentera le personnage.

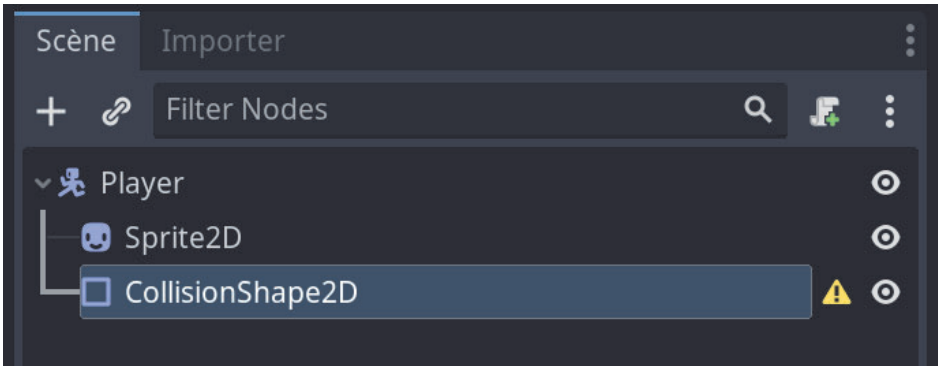
Parmi les ressources que je vous ai données, dans le dossier `Assets Jeu 1/Players/Player Blue`, vous trouverez différentes textures du personnage que nous allons créer. Utilisez l'image `PlayerBlue_Stand.png` comme texture par défaut lorsque le personnage se tient debout. Pour appliquer la texture à votre Sprite, glissez/déposez-la dans la variable TEXTURE du Sprite via l'inspecteur. Votre personnage devrait alors être visible comme sur la Figure 8.1.

Figure 8.1 : *La texture de notre personnage*



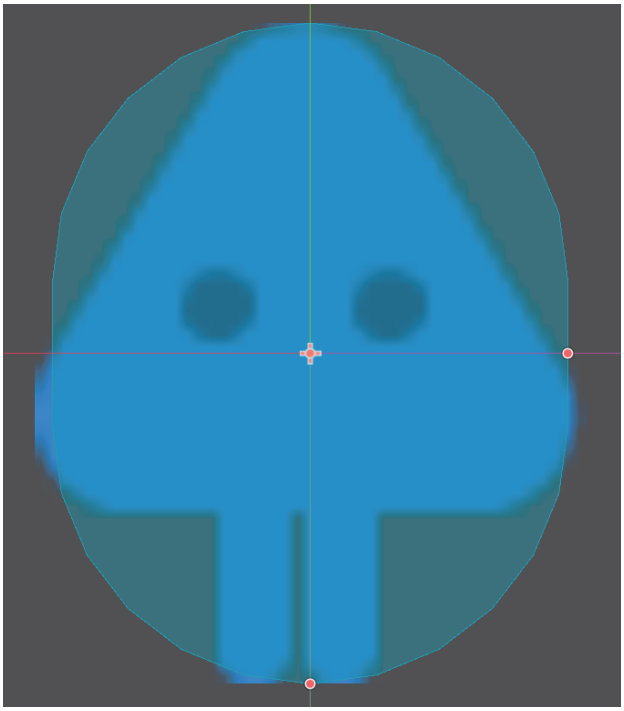
Nous ajoutons ensuite un CollisionShape2D à notre Player pour le rendre *solide*. Voici à quoi devrait ressembler votre scène :

Figure 8.2 : *Notre scène Player*



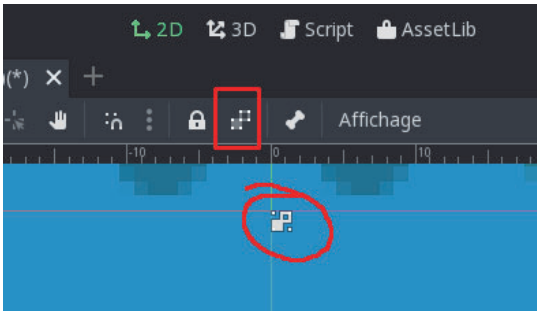
Il nous reste encore à ajouter un Shape à notre CollisionShape2D. Cliquez sur la propriété SHAPE dans l'inspecteur et choisissez une CapsuleShape2D. Une fois ceci fait, ajustez la taille de la capsule pour englober votre personnage.

Figure 8.3 : *Ajout d'un Shape au personnage*



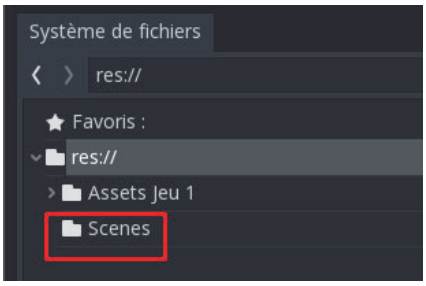
Vous pouvez maintenant masquer le CollisionShape2D en cliquant sur l'icône de l'œil dans la scène. Pensez également à sélectionner votre Player et à bloquer la sélection des enfants en utilisant l'outil de verrouillage. Lorsque la sélection des enfants est désactivée, vous constatez que l'icône apparaît sur le personnage :

Figure 8.4 : *Bloquer la sélection des enfants*



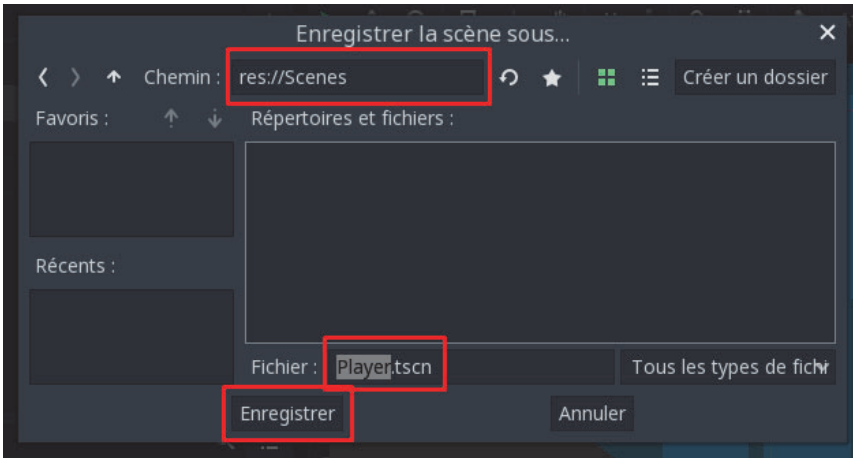
Nous allons maintenant sauvegarder notre travail. Pour cela, créez un nouveau dossier que vous appellerez Scenes.

Figure 8.5 : *Création d'un dossier*



Enregistrez ensuite votre scène dans ce dossier.

Figure 8.6 : *Sauvegarde du personnage*

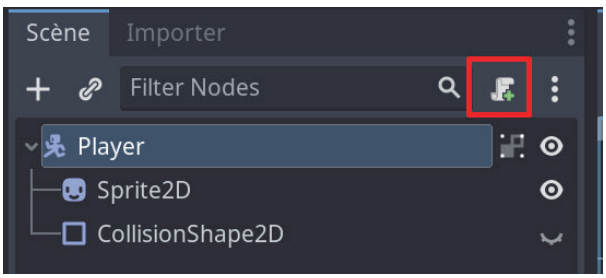


8.2. Création du script du personnage

Nous allons maintenant créer le script du personnage que nous ferons petit à petit évoluer. Ce script permettra notamment au joueur de contrôler le personnage.

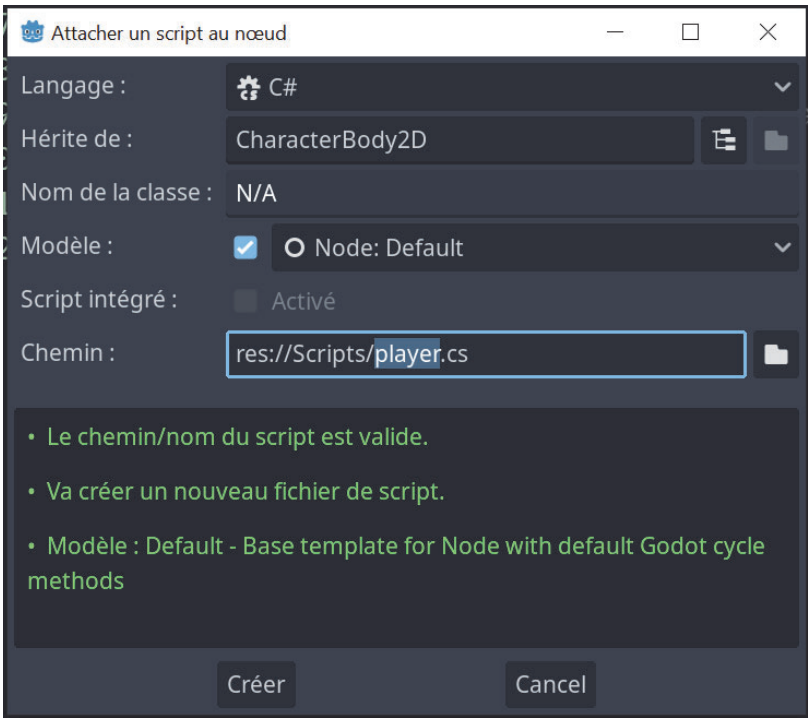
Sélectionnez votre personnage et créez un nouveau script en cliquant sur l'icône ATTACHER UN NOUVEAU SCRIPT.

Figure 8.7 : *Attacher un nouveau script*



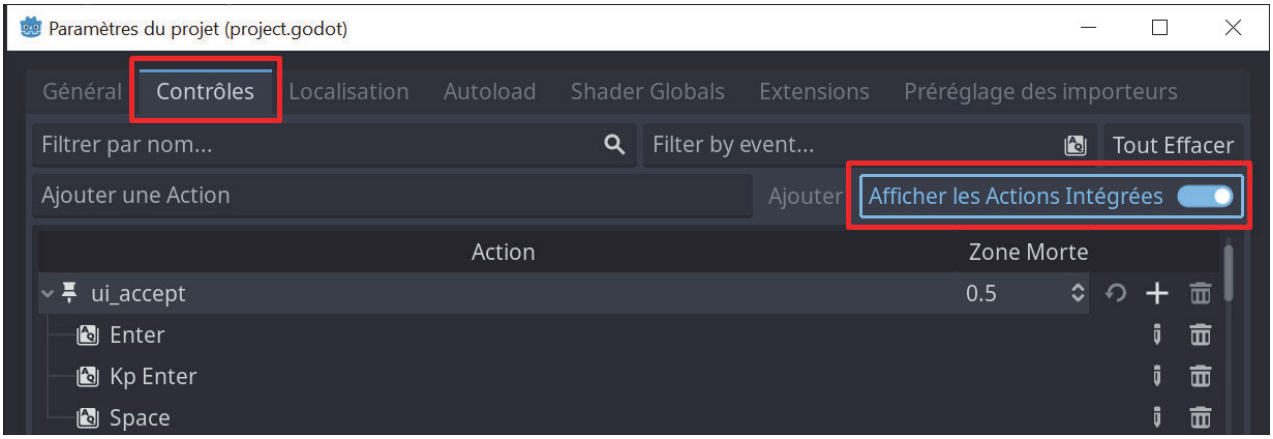
Conservez le nom par défaut, à savoir `Player.cs`. L'extension `.cs` est utilisée pour les scripts C# (c-sharp). Comme modèle de script, choisissez NODE: DEFAULT et sauvegardez le nouveau script dans un dossier `Scripts` que vous aurez créé dans votre système de fichiers.

Figure 8.8 : *Création d'un script dans un dossier*



Dans ce script, nous allons coder les mouvements du personnage. Pour cela, nous aurons besoin de configurer les inputs, c'est-à-dire les touches du clavier qui seront utilisées par le jeu. Pour configurer ces touches, nous devons nous rendre dans le menu PROJET/PARAMÈTRES DU PROJET. La fenêtre de paramètres s'ouvre alors. Cliquez sur l'onglet CONTRÔLES. Ici, commencez par activer l'option AFFICHER LES ACTIONS INTÉGRÉES qui permet d'afficher les inputs par défaut (voir Figure 8.9).

Figure 8.9 : *Affichage des contrôles et des actions*



Nous avons ensuite la possibilité d'associer des touches du clavier à des mots clés et de tester ces inputs dans le code. Nous allons associer les touches Z, Q, S et D aux actions `ui_up`, `ui_left`, `ui_down` et `ui_right`.

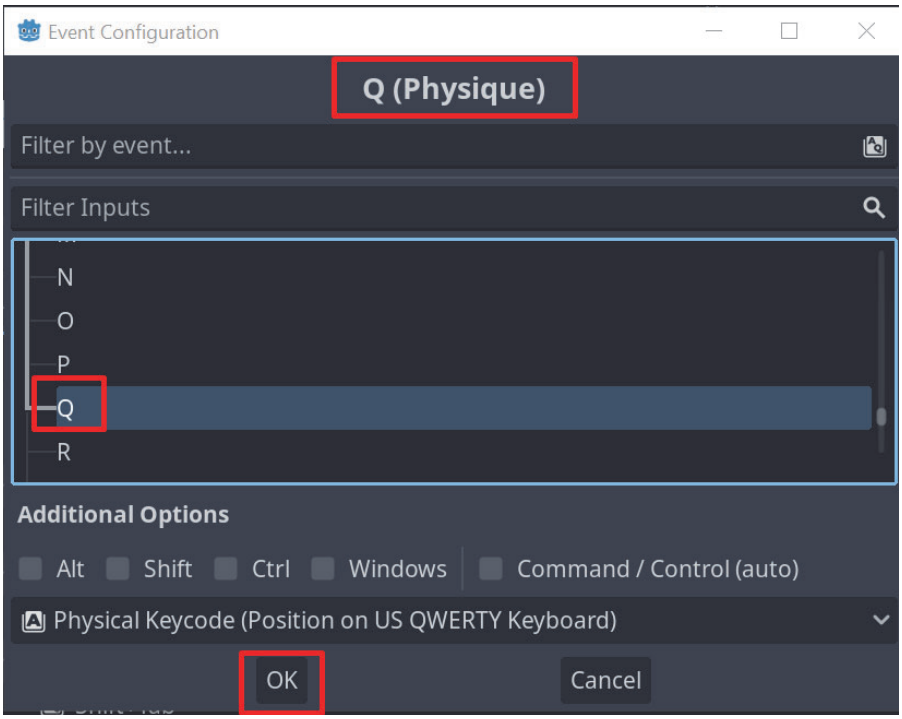
Note › *Dans le cadre de notre jeu, le personnage ne se déplacera que de gauche à droite donc seules les touches `Q` et `D` seront utilisées. Nous allons quand même configurer les touches `Z` et `S`, ainsi vous pourrez rapidement les mettre en place si besoin.*

Commençons par configurer `ui_left`. Recherchez cette entrée dans la liste des actions, cliquez sur le + à droite de la fenêtre et choisissez TOUCHE pour associer une touche à cette action.

Figure 8.10 : *Ajout d'une touche*

Une pop-up s'ouvre et vous permet de rechercher la touche à associer à l'action. Recherchez la touche Q qui correspond à la direction gauche et validez en cliquant sur OK.

Figure 8.11 : *Attribution de la touche*



Faites de même pour les trois autres actions à configurer.

Figure 8.12 : *Configuration des touches*

Une fois que vos touches sont configurées, fermez la fenêtre puis retournez sur votre script.

La première chose que nous allons faire, c'est créer une constante qui stockera la vitesse de déplacement du personnage. Cette constante sera nommée `SPEED` et aura une valeur de 300. Cela signifie que notre personnage pourra avancer à une vitesse de 300 pixels par seconde. Vous pourrez adapter cette valeur et les prochaines valeurs à vos besoins et vos envies.

```
// Constantes
public const float SPEED = 300.0f;
```

Note › *Lorsque la valeur est un float (nombre à virgule), nous devons ajouter la lettre `f` juste après ce nombre. Il s'agit d'une particularité de ce type de nombre.*

Supprimez ensuite les fonctions `_Ready` et `_Process` car nous n'en aurons pas besoin. Utilisez à la place la fonction `_PhysicsProcess`. Cette fonction est très similaire à `_Process` que nous avons déjà vue, mais elle est adaptée et conseillée lorsqu'on travaille avec un objet physique comme le CharacterBody2D de notre personnage.

Dans cette fonction, créez une variable de type `Vector2` (que vous nommerez par exemple `velocity`) qui permettra de définir le déplacement (vitesse/direction) du personnage. Cette variable aura pour valeur `Velocity` (avec un V majuscule) qui correspond à la vélocité actuelle du personnage.

```
// Fonction qui tourne en boucle
public override void _PhysicsProcess(double delta)
{
  Vector2 velocity = Velocity;
}
```

Attention › *Veillez à bien respecter les majuscules et minuscules en recopiant le code.*

GESTION DE LA PHYSIQUE DU JEU

Nous avons déjà parlé de la fonction `_Process`. Cette fonction tourne en boucle et permet de faire des choses comme vérifier en permanence les interactions clavier/souris par exemple. Dans cette section, nous allons travailler avec le moteur physique de Godot. Dès lors qu'il s'agit de physique, nous devons utiliser la fonction `_PhysicsProcess` au lieu de `_Process`. Le comportement de ces deux fonctions est sensiblement le même mais `_PhysicsProcess` est beaucoup plus adaptée lorsque

vous mettez à contribution le moteur physique. Je ne rentre pas dans les détails techniques car cela est complexe mais sachez juste que la fonction `_Process` n'est pas assez précise et des erreurs/bugs peuvent se produire lorsqu'il s'agit de calculs physiques.

C'est dans cette fonction qui tourne en permanence que nous testerons les inputs (c'est-à-dire que nous vérifierons si le joueur touche à son clavier) et c'est ici que nous appliquerons le mouvement.

Pour tester si le joueur appuie sur une touche du clavier, nous allons utiliser la fonction `Input.GetVector()`.

```
Input.GetVector();
```

DÉTECTER SI LE JOUEUR APPUIE SUR UNE TOUCHE DE SON CLAVIER

De façon générale, pour identifier quelle touche a été appuyée, il faut utiliser la fonction `IsActionPressed` avec en paramètre le nom de l'action qui lui est associée de la manière suivante. Adaptez simplement le nom de l'action testée pour détecter l'appui sur une autre touche.

```
if (Input.IsActionPressed("ui_right"))
{
  // Action à réaliser si le joueur a appuyé sur la flèche de droite
}
```

Cependant, je vous ai dit que dans notre cas nous utiliserons la fonction `GetVector()`. La raison est simple : les actions de direction (gauche/droite ou haut/bas) impliquent des vecteurs, d'où l'utilisation de `GetVector`. En effet, une direction est représentée par un vecteur. Dans ce cas particulier, il est plus simple et plus optimisé d'utiliser `GetVector`.

La fonction `GetVector()` retourne comme son nom l'indique, un vecteur représentant la direction de déplacement souhaitée. Il faut donc stocker ce vecteur dans une variable afin que nous soyons en mesure de l'utiliser le moment venu. Nous allons donc créer une variable `direction` de type `Vector2` et y stocker la valeur récupérée sur les quatre actions (haut, bas, gauche, droite) que nous avons configurées :

```
// Pour stocker le vecteur de direction
Vector2 direction = Input.GetVector(
  "ui_left", "ui_right", "ui_up", "ui_down"
);
```

Une fois ceci fait, nous devons écrire une condition qui permettra de vérifier si la direction est non nulle. Ce test permettra de savoir si le joueur a appuyé sur une touche de direction et si un mouvement doit être calculé.

```
// Test de la direction
if (direction != Vector2.Zero)
{

}
else
{

}
```

`if` permet de mettre en place la condition. `if (direction != Vector2.Zero)` signifie *Si la direction n'est pas nulle*. C'est le point d'exclamation qui permet de tester la négation (il indique "n'est pas égal à"). Le mot clé `else` permet quant à lui de décrire le comportement alternatif si la condition précédente n'est pas satisfaite.

Une fois que vous avez écrit votre condition, vous devez programmer ce qui doit se produire lorsqu'elle est respectée. Dans notre cas, nous voulons faire avancer le personnage vers la direction demandée.

Pour faire avancer le personnage sur l'axe X (de gauche à droite), il faut multiplier la direction sur l'axe X par notre constante `SPEED` et stocker le résultat dans la variable `velocity` créée plus haut :

```
velocity.X = direction.X * SPEED;
```

Note › *Notre jeu sera un jeu en vue de côté, le personnage ne pourra se déplacer que de gauche à droite. Les touches Z et S n'auront donc aucun impact sur ce bout de code. Si vous prévoyez de faire un jeu en vue de dessus, alors il vous suffira de procéder de la même façon sur l'axe Y pour pouvoir vous déplacer dans les quatre directions.*

Attention › *Écrivez bien `velocity.X` et `direction.X` avec un X majuscule, sinon cela ne fonctionnera pas. Il est également possible d'écrire `direction[0]` (0 correspondant à la valeur de X et 1 à la valeur de Y) pour modifier ou accéder à la valeur de l'axe X.*

Dans le `else`, pour que le mouvement prenne fin si le joueur ne touche pas à son clavier, nous devons faire en sorte que le personnage cesse de se déplacer. Pour mettre fin

au mouvement de façon naturelle et progressive (le personnage ne doit pas s'arrêter brusquement), nous utiliserons le bout de code suivant :

```
velocity.X = Mathf.MoveToward(Velocity.X, 0, Speed);
```

Note *› La fonction `MoveToward` permet de passer d'une valeur (dans notre cas la vélocité) initiale à une valeur cible (dans notre cas zéro) en un laps de temps (ici Speed). Cela permet de créer une transition lors de l'arrêt du personnage afin qu'il ne s'immobilise pas instantanément.*

Il nous reste maintenant à appliquer le mouvement au personnage. Pour cela, nous utiliserons la fonction `MoveAndSlide` qui permet de déplacer un corps suivant un vecteur (velocity). Cette fonction tient compte des collisions, donc le personnage sera bloqué par les murs dans son déplacement.

En dehors de la condition, attribuez la nouvelle valeur de `Velocity` et appelez la fonction `MoveAndSlide` de cette façon :

```
// On applique le mouvement
Velocity = velocity;
MoveAndSlide();
```

`MoveAndSlide` prend automatiquement en compte la valeur de `Velocity` pour appliquer le mouvement désiré.

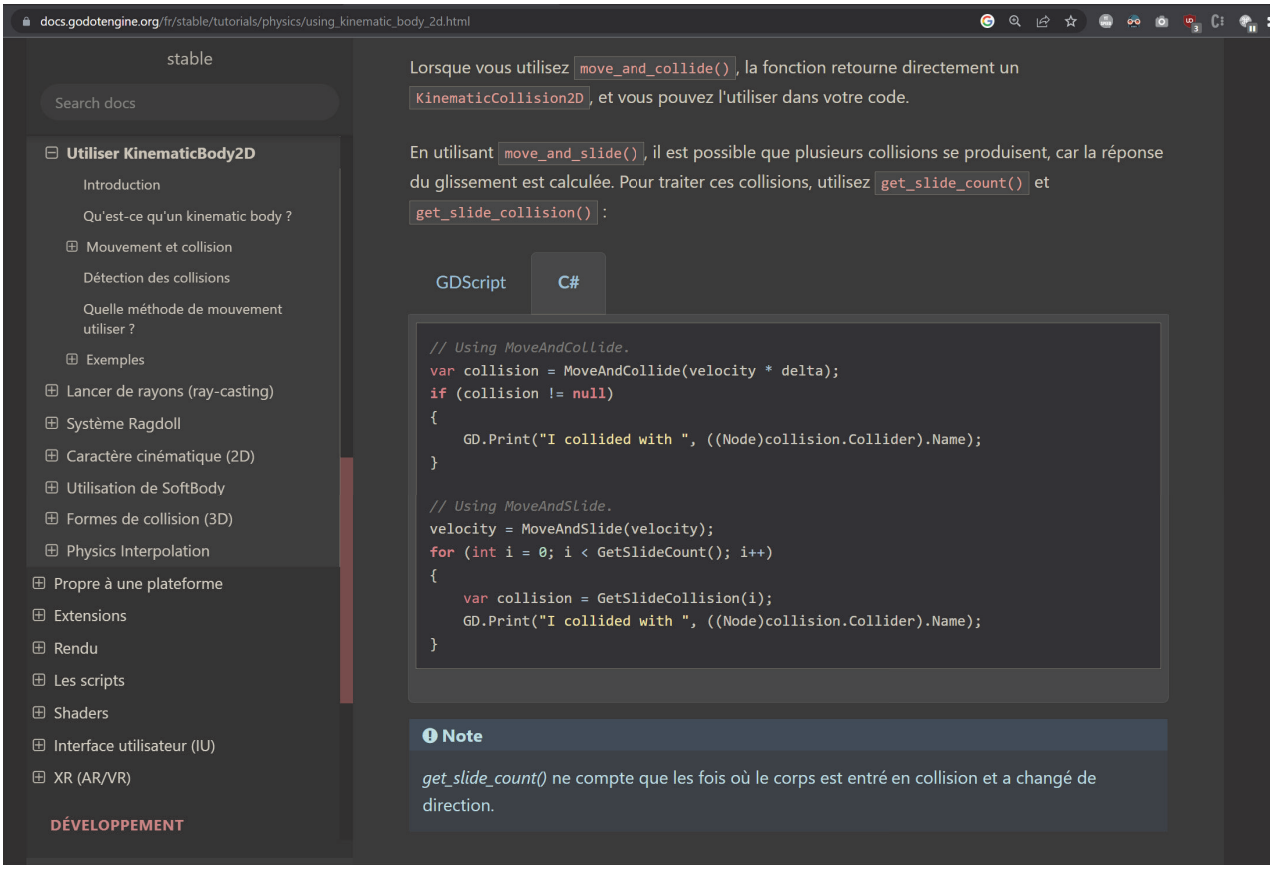
Si vous avez bien tout suivi, le script complet devrait ressembler à cela :

```
using Godot;
using System;

public partial class player : CharacterBody2D
{
  // Constantes
  public const float SPEED = 300.0f;

  // Fonction qui tourne en boucle
  public override void _PhysicsProcess(double delta)
  {
    // Vélocité du personnage
    Vector2 velocity = Velocity;

    // Pour stocker le vecteur de direction
    Vector2 direction = Input.GetVector(
      "ui_left", "ui_right", "ui_up", "ui_down"
    );

    // Test de la direction
    if (direction != Vector2.Zero)
```

```
        {
            velocity.X = direction.x * SPEED;
        }
        else
        {
            velocity.X = Mathf.MoveToward(Velocity.X, 0, Speed);
        }

        // On applique le mouvement
        Velocity = velocity;
        MoveAndSlide();
    }
}
```

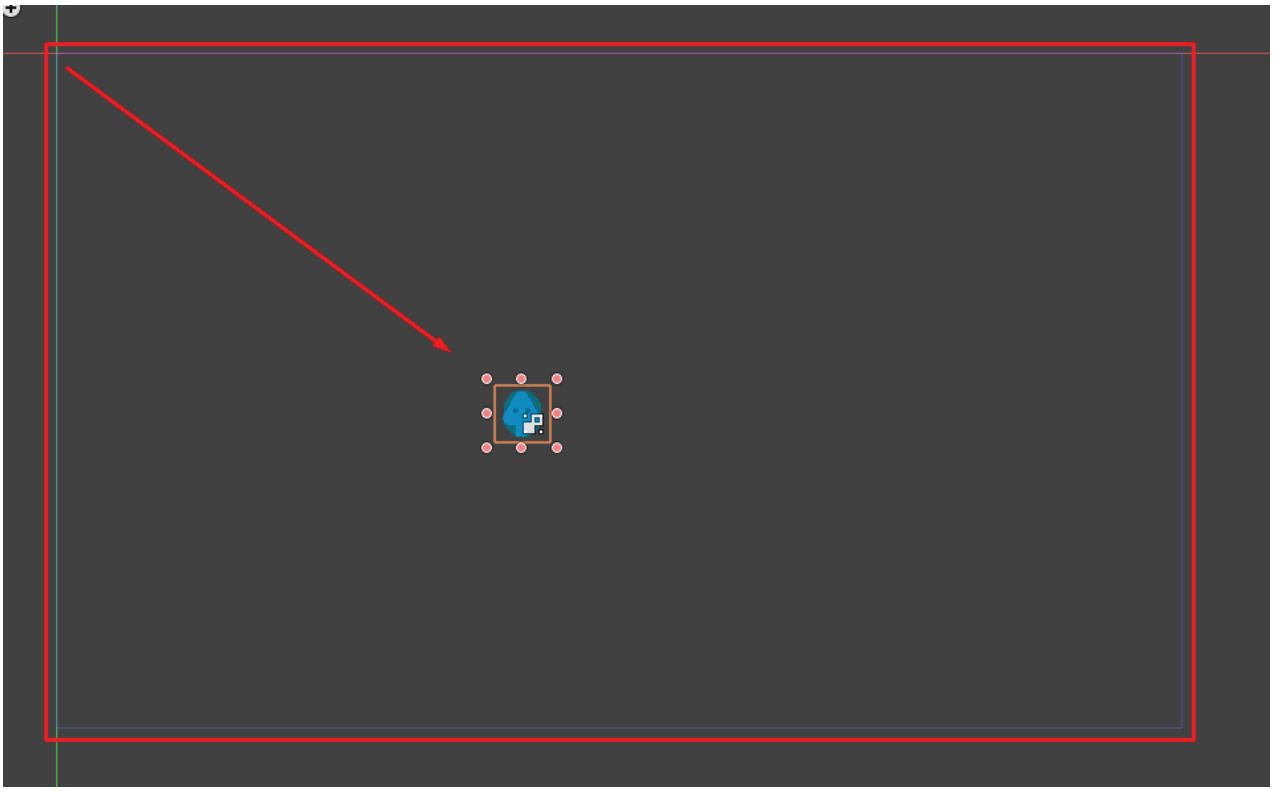
Ce code est suffisant pour faire bouger notre personnage, enregistrez-le. Si vous avez besoin d'en savoir plus sur le fonctionnement d'une fonction comme par exemple `MoveAndSlide`, n'hésitez pas à vous rendre sur la documentation officielle de Godot.

Figure 8.13 : *`MoveAndSlide` expliqué dans la documentation officielle de Godot*



Avant de tester notre code, nous allons déplacer notre personnage et le positionner au centre de l'écran pour qu'il soit bien visible. Pour cela, retournez dans la vue 2D, sélectionnez le personnage et déplacez-le pour le positionner au centre de l'écran comme sur la Figure 8.14.

Figure 8.14 : *Placement du personnage au centre de l'écran*



Souvenez-vous que ce que verra le joueur c'est tout ce qui se trouve dans le carré coloré que vous voyez dans le viewport. Une fois que tout est prêt, lancez la scène avec F6. Votre jeu se lance et vous pouvez utiliser les flèches ou Q et D pour faire bouger votre personnage de gauche à droite.

Vous venez de développer la première brique du script de votre personnage. Si tout est bien configuré, le déplacement horizontal fonctionne. Pour le moment cela reste très basique, nous n'avons pas encore de gravité mais nous progressons. Nous nous occuperons de la gravité lorsque nous aurons un sol sur lequel tomber.

Dans le chapitre suivant, nous verrons comment animer notre personnage et donc comment utiliser les outils d'animation proposés par Godot.

9

Animation du personnage

Dans 99,9 % des jeux vidéo, les personnages sont animés. Les animations peuvent être en 3D si nos personnages sont modélisés en 3D (dans ce cas-là, les animations sont basées sur un squelette et des articulations) ou elles peuvent être en 2D. En 2D, il existe principalement deux façons de gérer les animations :

- Soit vous avez un personnage avec toutes les parties du corps découpées et animées indépendamment, c'est le cas par exemple de Rayman :

Figure 9.1 : Rayman (Ubisoft)

- Soit vos animations sont basées sur des feuilles de sprites (*Sprite Sheet*) comme c'est le cas pour beaucoup de jeux 2D. Cette solution consiste à dessiner notre personnage plusieurs fois afin de mettre côte à côte une série d'images formant une animation (voir Figure 9.2).

C'est cette dernière solution qui sera utilisée pour notre projet. Elle est simple à mettre en place et automatiquement gérée par Godot et son outil d'animation. Vous trouverez toutes les animations des personnages que nous allons utiliser parmi les ressources fournies avec le livre.

Figure 9.2 : *Sprite Sheet*

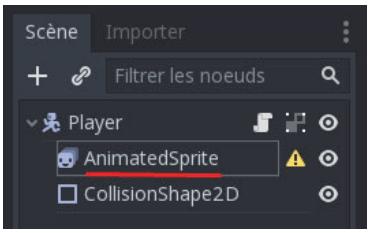
Le principe est très simple : si une animation de marche est composée de cinq images, nous devons les interchanger pour animer le personnage. Nous pouvons par exemple cadencer l'animation à dix images par seconde et l'animation prendra vie.

Nous allons appliquer ce procédé en créant une animation de marche pour notre personnage. Nous jouerons cette animation lorsqu'il se déplacera de gauche à droite. Lorsque le joueur relâche les touches du clavier, le personnage aura une animation d'attente (*idle*) basique.

9.1. Création des animations

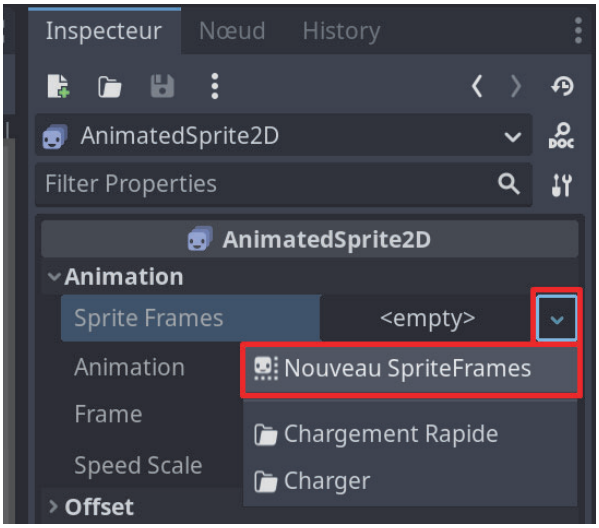
Actuellement, notre personnage est visible grâce au composant Sprite2D. Ce composant ne permet pas de gérer des animations. Nous allons devoir le remplacer par un AnimatedSprite2D. Ce nouveau composant est conçu pour gérer les animations des sprites 2D. Pour remplacer votre Sprite2D par un AnimatedSprite2D, cliquez droit sur votre Sprite et sélectionnez CHANGER DE TYPE. Recherchez l'AnimatedSprite2D et validez. Pour éviter toute confusion, double-cliquez sur le nouveau composant créé et renommez-le `AnimatedSprite`.

Figure 9.3 : *AnimatedSprite2D*



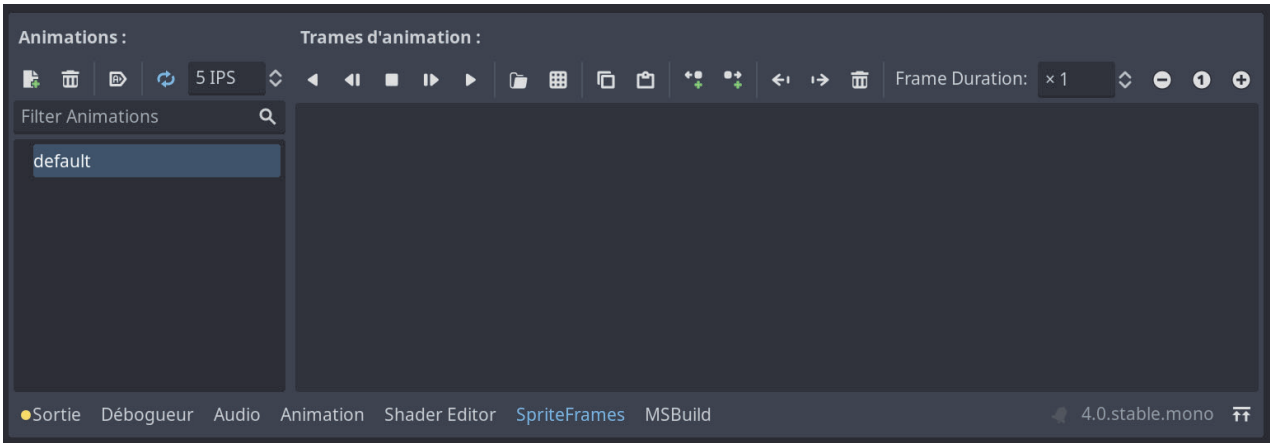
Vous constatez qu'un avertissement apparaît sur votre AnimatedSprite. On nous demande de créer une ressource de type SpriteFrames pour qu'une image puisse être affichée à l'écran. Nous allons faire cela immédiatement. Dans l'inspecteur, à la section ANIMATION, vous trouverez la propriété SPRITE FRAMES, cliquez dessus et choisissez NOUVEAU SPRITEFRAMES :

Figure 9.4 : *Création du SpriteFrames*



Une fois que vous avez créé le SpriteFrames, cliquez de nouveau dessus pour ouvrir la fenêtre associée. C'est là que vous pouvez glisser les textures que vous voulez utiliser pour vos animations.

Figure 9.5 : *La fenêtre d'animation*



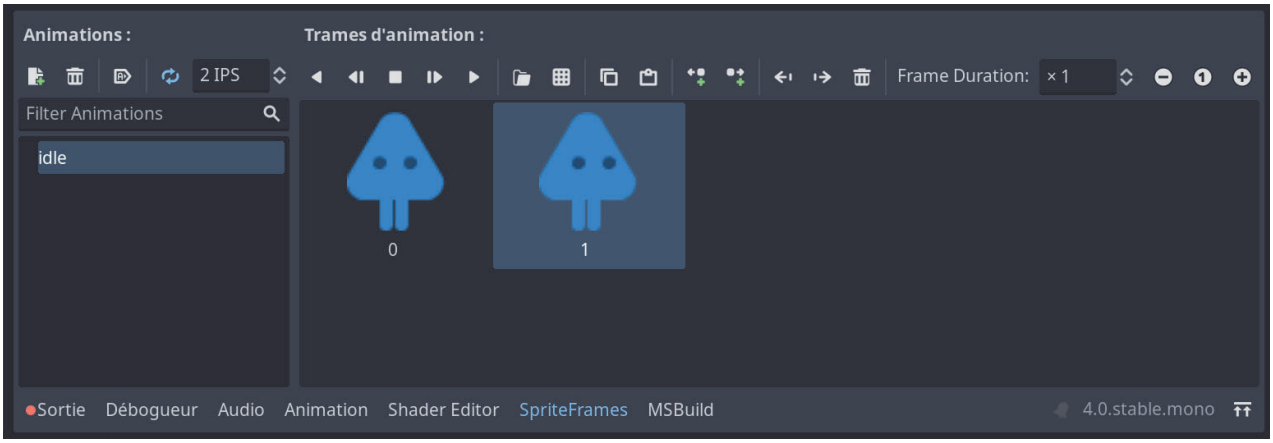
Une animation nommée `default` est présente par défaut. Renommez-la `idle`. Nous allons lui associer les textures de notre animation.

Sachez que notre personnage ne dispose pas d'animation *idle*, nous n'avons à disposition que la texture `Stand` [Tenir debout]. Nous pouvons donc créer une animation qui

sera... statique ! Cependant, nous n'allons pas faire cela. Afin d'avoir un semblant d'animation avec un tout petit mouvement, nous ajouterons la texture `walk1`, qui est similaire à `Stand`, mais avec un très léger décalage au niveau des yeux.

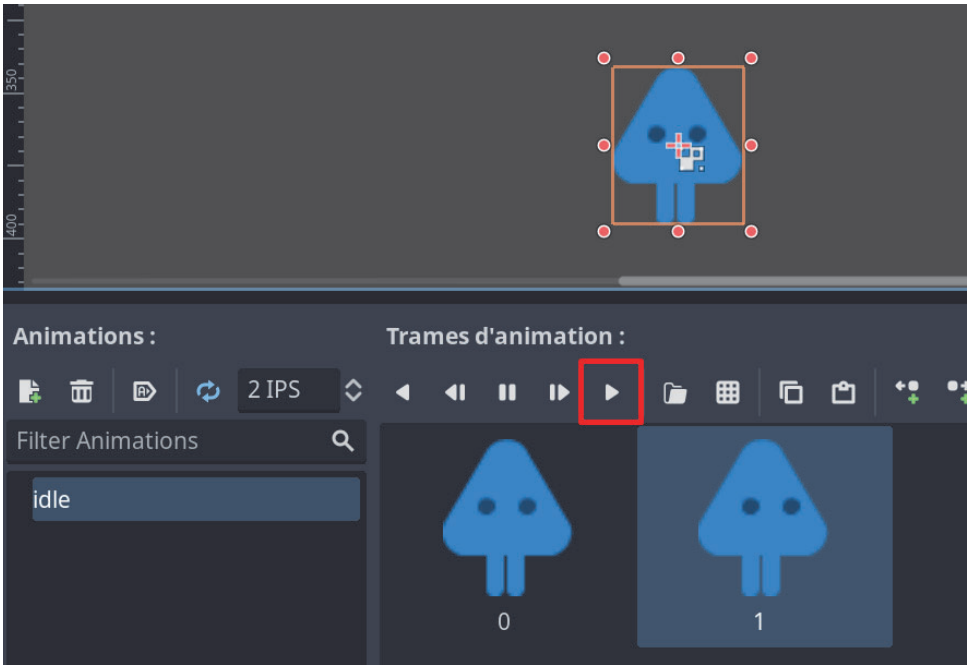
Faites donc glisser ces deux textures dans la zone principale de l'animation. Vous pouvez voir le paramètre VITESSE (FPS) dans la partie de gauche. La vitesse par défaut est de 5. Mettez une vitesse de 2 pour cette animation. Cela signifie que les textures s'interchangeront deux fois par seconde.

Figure 9.6 : *Animation idle*



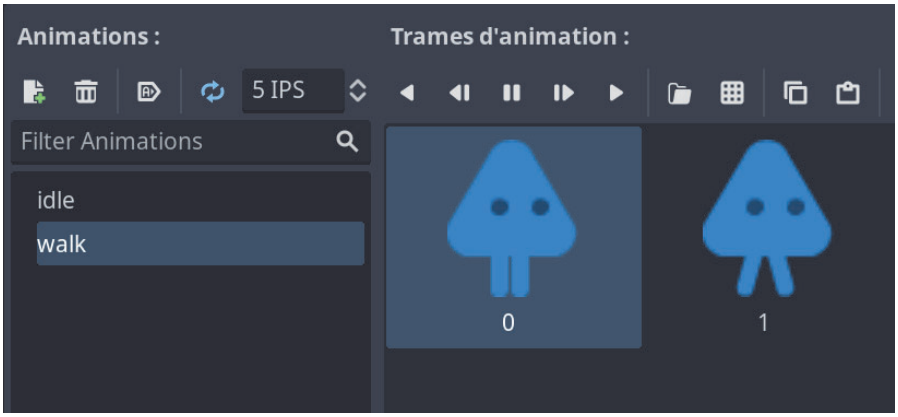
Vous pouvez vérifier que votre animation est bien configurée. Pour cela, cliquez sur l'icône PLAY au niveau de la fenêtre ANIMATIONS afin de lancer l'animation.

Figure 9.7 : *Tester l'animation dans le viewport*



Si tout est bien configuré, vous devriez voir les yeux du personnage bouger. Maintenant que vous avez compris le principe, nous allons passer à une animation un peu plus intéressante : l'animation de marche ! Toujours dans l'outil d'animation, créez une nouvelle animation en cliquant sur le bouton dédié en haut à gauche de la fenêtre d'animation. Appelez cette animation `walk` et ajoutez les textures `walk1` et `walk2`. Laissez la vitesse à 5.

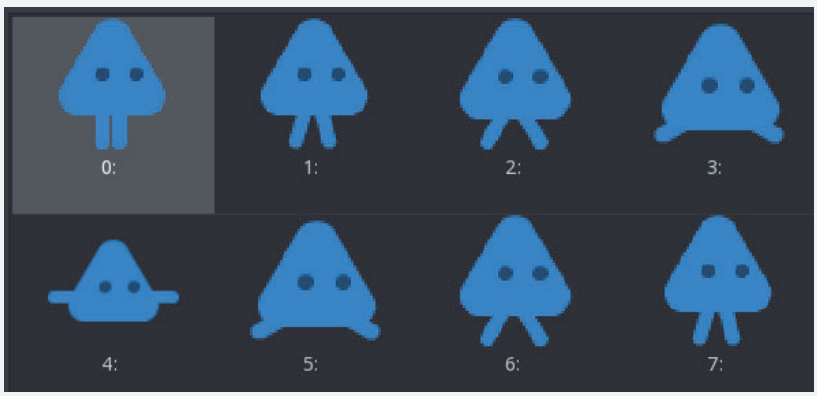
Figure 9.8 : *Animation walk*



ANIMATIONS PLUS COMPLEXES

Dans ce pack d'assets, le personnage bleu que nous utilisons dispose de cinq animations de marche. Ici je n'utilise que deux images car cela me convient mais vous pouvez utiliser les cinq images si vous voulez une animation différente ou plus complète. Vous pouvez également utiliser les images 1, 2, 3, 4, 5 et réutiliser les images 4, 3, 2 pour créer une sorte de boucle :

Figure 9.9 : *Exemple d'animation plus complexe*

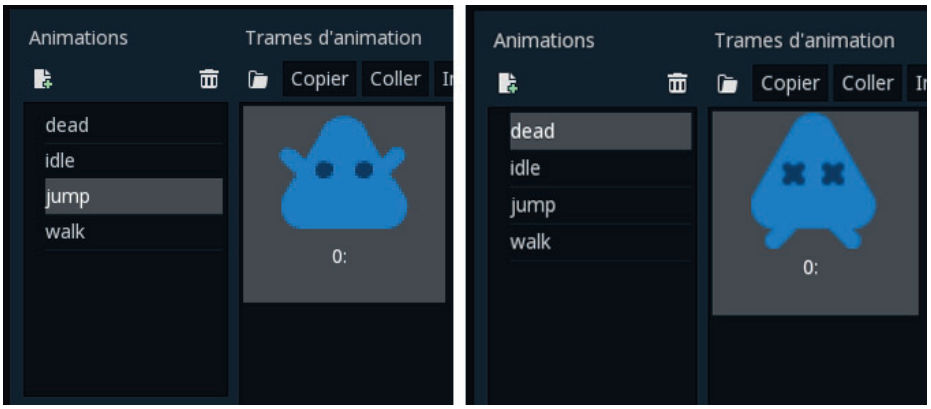


Par simplicité, je resterai sur l'animation à deux frames.

Testez votre animation dans le viewport à l'aide du bouton PLAY. Comme le personnage dispose de plus d'une animation, il vous faudra sélectionner celle que vous voulez lancer via la liste.

Pour prendre un peu d'avance sur la suite, profitons que nous sommes sur les animations pour créer deux nouvelles animations. Créez les animations `jump` et `dead` et utilisez les sprites `up3` pour le saut (*jump*) et `dead` pour le personnage KO (*dead*).

Figure 9.10 : *Autres animations*



Nous avons désormais un ensemble de quatre animations que nous pourrons utiliser dans notre jeu.

9.2. Programmation des animations

Maintenant que nous avons des animations à disposition, nous allons les utiliser pour animer notre personnage de façon adéquate en fonction des mouvements effectués. Nous allons faire marcher le personnage lorsqu'il se déplace et le mettre en *idle* quand le mouvement s'arrête.

Retournez dans le script `Player.cs` afin de l'éditer. Nous avons vu dans la première partie de ce manuel, en introduction au langage C#, qu'il était possible d'accéder à un nœud enfant grâce à la fonction `GetNode`. Or, nous avons besoin d'accéder à l'AnimatedSprite pour lancer l'animation.

Créez une nouvelle variable privée de type AnimatedSprite2D qui permettra de stocker l'AnimatedSprite :

```
private AnimatedSprite2D AnimatedSprite;
```

Ensuite, dans la fonction `_Ready`, utilisez `GetNode` afin de récupérer ce composant AnimatedSprite2D et de l'attribuer à notre variable :

```
public override void _Ready()
{
  AnimatedSprite = GetNode<AnimatedSprite2D>("AnimatedSprite2D");
}
```

Pour lancer une animation, vous devez utiliser la fonction `Play` du AnimatedSprite. Cette fonction `Play` prend en paramètre le nom de l'animation à jouer. Dans notre exemple, nous voulons jouer l'animation de marche (`walk`), nous écrirons donc :

```
AnimatedSprite.Play("walk");
```

Ce code doit être écrit dans le `if (direction != Vector2.Zero)` pour que l'animation se déclenche lorsque le personnage se déplace. Nous devons aussi lancer l'animation `idle` lorsque l'utilisateur retire ses doigts du clavier. Dans le `else`, vous devez écrire :

```
AnimatedSprite.Play("idle");
```

Vous devriez avoir le script suivant pour votre fonction `_PhysicsProcess` :

```
public override void _PhysicsProcess(double delta)
{
  // Vélocité du personnage
  Vector2 velocity = Velocity;

  // Pour stocker le vecteur de direction
  Vector2 direction =
 Input.GetVector("ui_left", "ui_right", "ui_up", "ui_down");

  // Test de la direction
  if (direction != Vector2.Zero)
  {
    velocity.X = direction.X * SPEED;
    AnimatedSprite.Play("walk");
  }
  else
  {
    velocity.X = Mathf.MoveToward(Velocity.X, 0, SPEED);
    AnimatedSprite.Play("idle");
  }

  // On applique le mouvement
  Velocity = velocity;
  MoveAndSlide();
}
```

Vous pouvez tester votre programme et constater que les animations fonctionnent comme convenu !

Il y a une petite subtilité à prendre en compte. Dans notre exemple, le personnage regarde toujours du même côté (vers la droite) même lorsqu'il se déplace vers la gauche. Ce n'est pas bon, il faut orienter le personnage dans la direction vers laquelle il regarde. Lorsque le personnage va à gauche, il doit regarder à gauche et s'il va à droite il doit regarder vers la droite. Pour faire cela, nous utiliserons la propriété `FlipH` du AnimatedSprite qui permet de faire une symétrie horizontale de notre Sprite. Nous devons donc vérifier si la direction est positive ou négative (droite ou gauche) et faire la symétrie si nécessaire. Dans notre cas, le code serait le suivant :

```
public override void _PhysicsProcess(double delta)
{
  Vector2 velocity = Velocity;

  Vector2 direction =
 Input.GetVector("ui_left", "ui_right", "ui_up", "ui_down");

  if (direction != Vector2.Zero)
  {
    velocity.X = direction.X * SPEED;
    AnimatedSprite.Play("walk");
    if (direction.X > 0) {
      AnimatedSprite.FlipH = false;
    }
    else
    {
      AnimatedSprite.FlipH = true;
    }
  }
  else
  {
    velocity.X = Mathf.MoveToward(Velocity.X, 0, SPEED);
    AnimatedSprite.Play("idle");
  }

  // On applique le mouvement
  Velocity = velocity;
  MoveAndSlide();
}
```

Si vous testez votre programme, vous verrez que le personnage s'oriente maintenant convenablement selon sa direction.

Maintenant que nous avons bien avancé la création de notre personnage, nous allons créer une plateforme afin de lui donner la possibilité de marcher sur un sol ! Cela nous permettra par la suite de travailler avec la physique.

10

Création d'une plateforme 2D

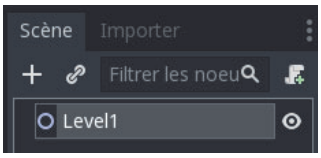
Dans ce chapitre, nous allons créer notre première plateforme 2D. Les plateformes seront utilisées pour créer le sol et les zones sur lesquelles le personnage pourra marcher. Pour commencer en douceur, nous allons nous concentrer sur une seule plateforme que nous utiliserons pour faire des tests. Une fois nos essais faits, nous verrons comment créer rapidement de nombreuses plateformes pour des niveaux de grande envergure.

10.1. Création d'une nouvelle scène

Actuellement, nous n'avons que la scène du joueur à notre disposition. Comme expliqué dans la première partie de ce livre, dans Godot, nous découpons au maximum les éléments du jeu en scènes. Cela signifie que le joueur est une scène à part entière. Et notre plateforme va constituer une autre scène indépendante.

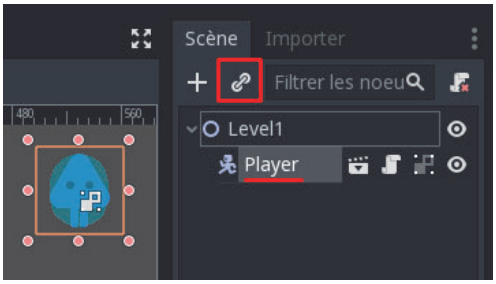
Nous allons donc créer une nouvelle scène (en cliquant sur le petit + au-dessus du viewport). Dans cette nouvelle scène, nous ajoutons un Node2D que nous allons renommer en `Level1`.

Figure 10.1 : *Nœud principal*



Une fois le nœud créé, pensez à sauvegarder votre scène `Level1` dans le dossier `Scenes` du projet. Dans cette nouvelle scène, nous allons instancier notre personnage `Player` :

Figure 10.2 : *Instanciation du Player*

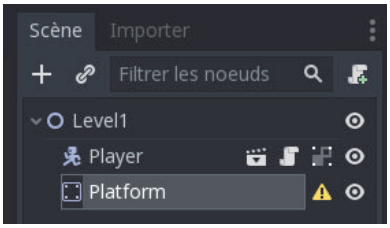


Le personnage nous permettra de tester notre scène et de mettre en place la gravité car il sera censé tomber sur la plateforme ou tomber dans le vide s'il dépasse les limites de la plateforme.

10.2. Création de la plateforme

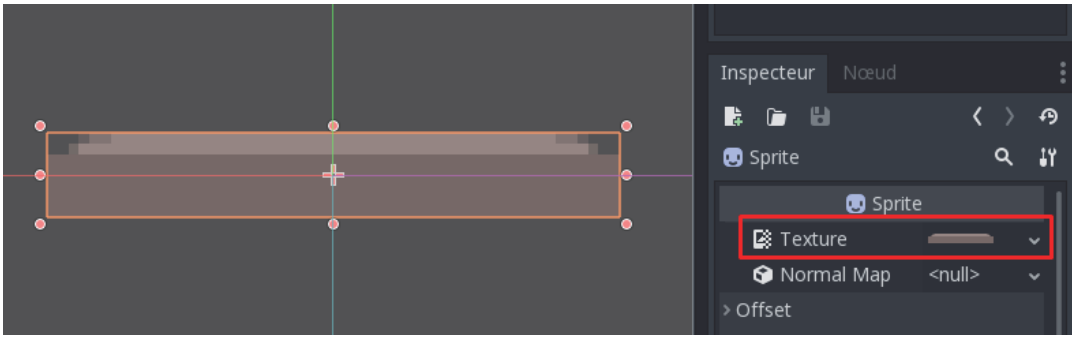
Nous allons maintenant créer une plateforme élémentaire qui nous permettra de coder la gravité. En effet, sans sol, nous n'avons pas de repère pour mettre en place la gravité. Cliquez sur votre nœud `Level1` et ajoutez-lui un StaticBody2D en tant qu'enfant. Renommez ce StaticBody2D en `Platform`.

Figure 10.3 : *Ajout du StaticBody2D*



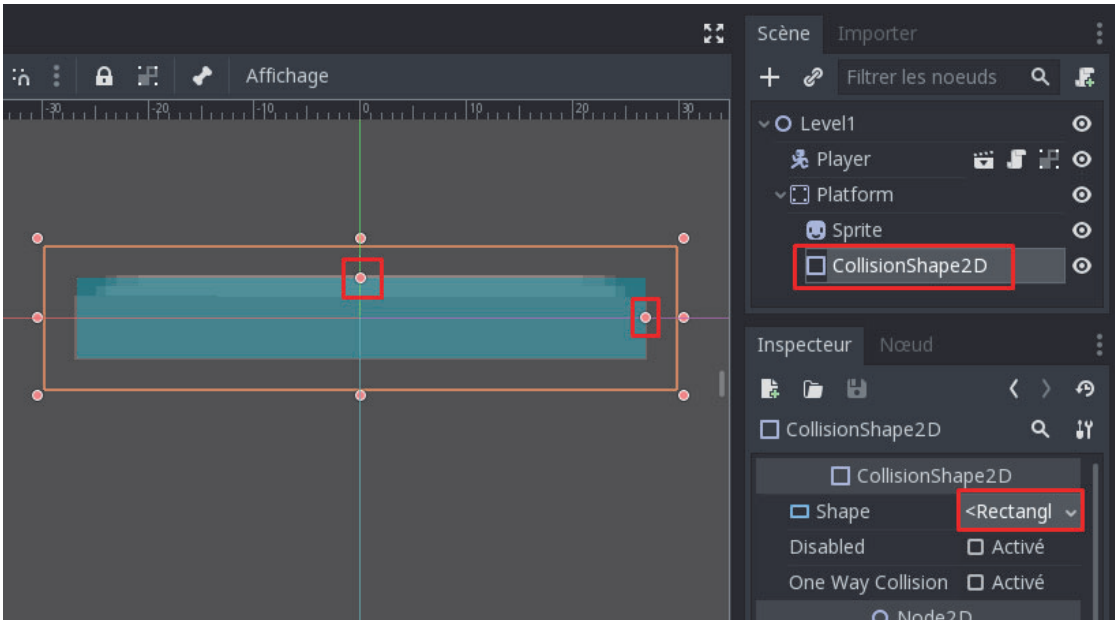
Nous devons maintenant ajouter un Sprite2D pour que la plateforme soit visible. Ajoutez donc ce Sprite en enfant du StaticBody. Au niveau de la texture, utilisez celle de votre choix, cela n'a pas beaucoup d'importance car nous n'utiliserons pas cette plateforme pour notre jeu, il s'agit pour le moment d'un simple test. Utilisez de préférence une texture allongée horizontalement afin de simuler une sorte de sol. Parmi les textures livrées avec le pack de ressources, vous retrouverez, dans le dossier `Other` une texture adaptée nommée `buttonFloor_pressed.png`.

Figure 10.4 : *Ajout d'une texture*



Pour terminer, nous devons ajouter un objet de collision sur notre plateforme. Ajoutez un CollisionShape2D qui sera un rectangle. Ajustez ce Shape à la forme de votre plateforme.

Figure 10.5 : *Ajout de la collision*

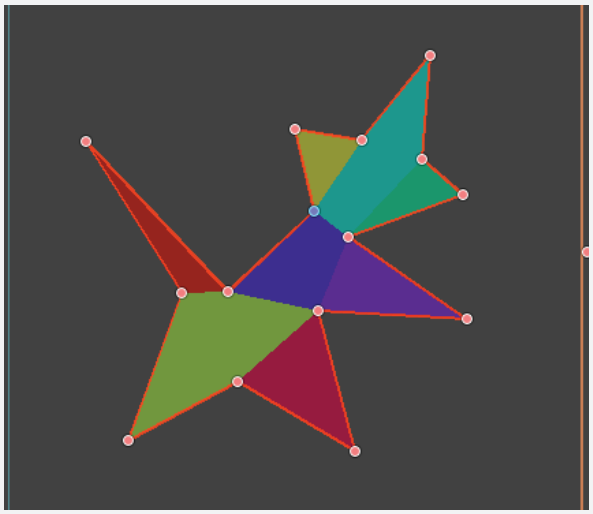


Vous pouvez masquer le Shape en cliquant sur l'œil de la fenêtre de scène. Pensez également à bloquer les enfants de la plateforme. Sauvegardez votre travail.

COLLISIONPOLYGON2D

Si vous avez une plateforme complexe ou une texture très particulière avec une forme plus évoluée qu'un simple rectangle, vous pouvez utiliser un CollisionPolygon2D qui vous permettra de dessiner à main levée votre contour selon la forme de votre objet de collision. Ce dessin qui est le contour de votre forme s'appelle un *PolygonShape*.

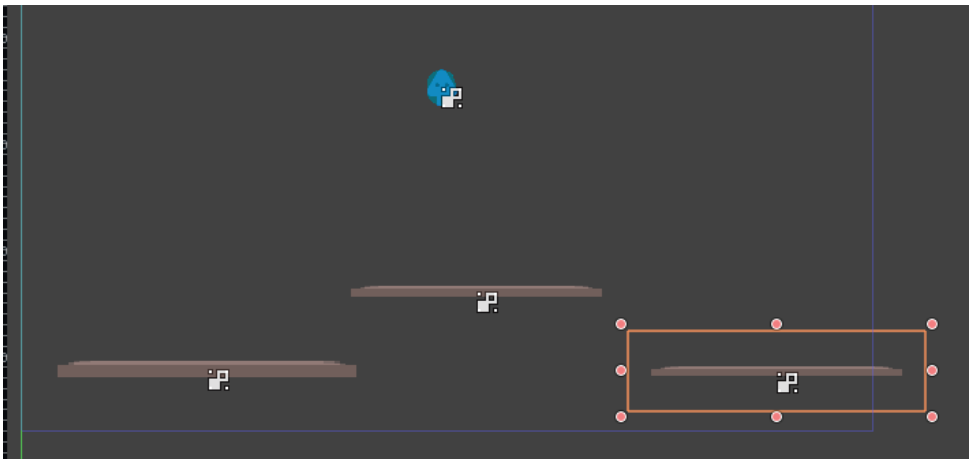
Figure 10.6 : *Exemple de PolygonShape*



Vous pouvez détourer votre Sprite en cliquant pour créer des points qui seront reliés pour former le Shape. Cela peut être très utile si vous avez besoin d'une grande précision dans vos collisions.

Maintenant que nous avons notre plateforme, nous allons pouvoir programmer la gravité qui s'appliquera sur notre personnage. Pour réaliser vos tests, vous pouvez agrandir et dupliquer la plateforme afin d'avoir des éléments sur lesquels marcher.

Figure 10.7 : *Duplication du sol*



10.3. Programmation de la gravité

Dans la première partie de ce livre, nous avons vu que la gravité était automatiquement active sur les RigidBody2D. Cela n'est pas le cas pour les CharacterBody2D comme notre personnage. Dans ce cas, nous devons programmer nous-mêmes la gravité.

Retournez dans le script du Player afin de modifier la fonction `_PhysicsProcess`. Si le personnage est dans les airs (s'il ne touche pas le sol), nous devons l'attirer vers le bas. Nous utiliserons la fonction `IsOnFloor()` pour savoir s'il touche le sol ou pas.

Note › *Pour déplacer le personnage, nous utilisons la fonction* `MoveAndSlide`*. Cette fonction a pour particularité de savoir où se trouve le sol grâce à sa normale (un vecteur). Sachant cela* `IsOnFloor` *est capable de connaître cette information pour déterminer si le personnage touche le sol ou pas.*

Quand le personnage est dans les airs, nous allons modifier le vecteur Y de `velocity` pour le faire descendre. Nous multiplierons ce vecteur par une valeur de gravité. Enfin, nous animerons le personnage avec l'animation de saut pendant qu'il est en hauteur.

Pour faire cela, dans la fonction `_PhysicsProcess`, en dessous de la déclaration de la variable `direction`, vous ajouterez la condition suivante :

```
// Pour appliquer la gravité quand on ne touche pas le sol
if (!IsOnFloor())
{
  velocity.Y += gravity * (float)delta;
  AnimatedSprite.Play("jump");
}
```

Note › *Afin que la gravité soit la même sur tous les ordinateurs, nous devons multiplier sa valeur par le delta. Comme delta est une variable de type double, nous devons mettre le type float entre parenthèses juste avant pour faire la conversion dans le bon type.*

Ce code ne peut pas marcher tel quel car la variable `gravity` n'existe pas encore. Au début de votre programme, en dessous des variables existantes, ajoutez-la comme suit :

```
// Récupération de la gravité depuis les options
public float gravity = ProjectSettings.GetSetting("physics/2d/
default_gravity").AsSingle();
```

Cette variable `gravity` récupèrera sa valeur depuis les paramètres du projet. Par défaut, une valeur de gravité existe dans les réglages du projet et la notation précédente permet de récupérer cette valeur à la volée.

Note › *Cette façon de récupérer la gravité permet de se baser sur les propriétés physiques définies dans le projet. On aurait pu préciser une valeur en dur à la place mais cela aurait été moins propre.*

Enfin, nous devons également utiliser la fonction `IsOnFloor()` dans des conditions placées au-dessus de chaque appel à la fonction `Play`. En effet, ces animations ne doivent se déclencher que si le personnage touche le sol. Le code du script `Player` est désormais le suivant :

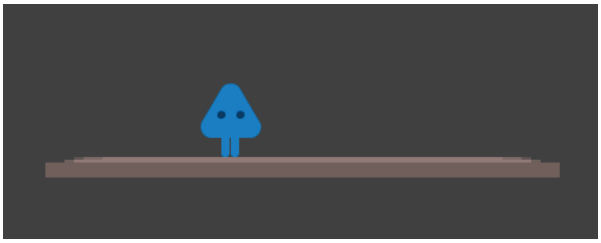
```
using Godot;
using System;

public partial class player : CharacterBody2D
{
  // Constantes
  public const float SPEED = 300.0f;
  private AnimatedSprite2D AnimatedSprite;
  // Récupération de la gravité depuis les options
  public float gravity = ProjectSettings.GetSetting("physics/2d/
default_gravity").AsSingle();

  public override void _Ready()
  {
    AnimatedSprite = GetNode<AnimatedSprite2D>("AnimatedSprite2D");
  }
```

```
  // Fonction qui tourne en boucle
  public override void _PhysicsProcess(double delta)
  {
    // Vélocité du personnage
    Vector2 velocity = Velocity;

    // Pour stocker le vecteur de direction
    Vector2 direction =
Input.GetVector("ui_left", "ui_right", "ui_up", "ui_down");

    // Pour appliquer la gravité quand on ne touche pas le sol
    if (!IsOnFloor())
    {
      velocity.Y += gravity * (float)delta;
      AnimatedSprite.Play("jump");
    }

    // Test de la direction
    if (direction != Vector2.Zero)
    {
      velocity.X = direction.X * SPEED;
      if (IsOnFloor())
      AnimatedSprite.Play("walk");
      if (direction.X > 0) {
        AnimatedSprite.FlipH = false;
      }
      else
      {
        AnimatedSprite.FlipH = true;
      }
    }
    else
    {
      velocity.X = Mathf.MoveToward(Velocity.X, 0, SPEED);
      if (IsOnFloor())
      AnimatedSprite.Play("idle");
    }


    // On applique le mouvement
    Velocity = velocity;
    MoveAndSlide();
  }
}
```

Note › *Lorsqu'une condition n'englobe qu'une seule et unique ligne de code, alors les accolades ne sont pas nécessaires. La condition sans accolades s'applique automatiquement à la ligne se trouvant juste en dessous de celle-ci.*

Désormais, votre personnage devrait tomber. Si lancez le jeu et que vous avez bien placé la plateforme en dessous de celui-ci, le personnage devrait se retrouver sur le sol.

Figure 10.8 : Test de la gravité



Vous avez réussi à mettre en place la gravité pour le personnage. Celui-ci n'est aspiré par le sol que lorsqu'il est dans les airs.

10.4. Programmation du saut

Pour conclure cette partie dédiée à la gravité, nous allons coder la fonction de saut qui permettra à notre personnage de faire un bond et de lutter contre cette gravité l'espace d'un instant.

Pour que notre personnage puisse sauter, il faudra passer une valeur négative à son `velocity.Y`. Nous devons donc définir une constante qui stockera la valeur de la puissance du saut. Une force de −600 me paraît bien dans notre cas selon mes tests. Vous pourrez par la suite ajuster toutes ces valeurs selon vos besoins.

```
public const float JumpVelocity = -600.0f;
```

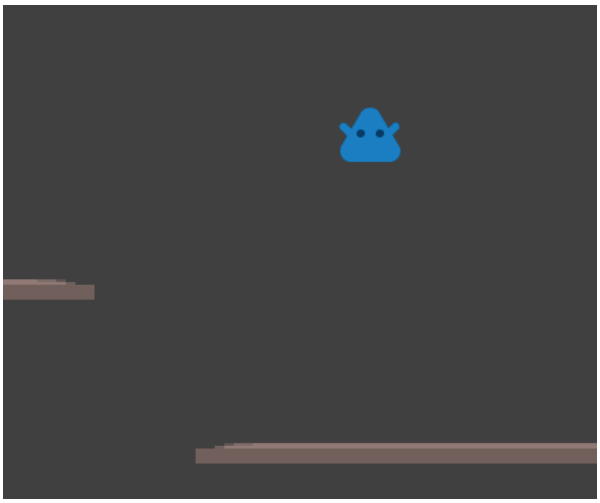
Nous allons maintenant créer la fonctionnalité de saut. Juste en dessous de la condition utilisée pour gérer la gravité, ajoutez une nouvelle condition. Cette dernière doit tester si le joueur appuie sur la barre d'espace et si le personnage est au sol. Le cas échéant, il peut sauter. Le code pour réaliser cela est le suivant :

```
// Gestion du saut si on appuie sur espace
if (Input.IsActionJustPressed("ui_accept") && IsOnFloor())
{
  velocity.Y = JumpVelocity;
}
```

Ce bout de code appliquera une force inverse à la gravité si le joueur appuie sur la barre d'espace et que le personnage est bien au sol au moment où la touche est appuyée.

Si vous testez le programme, votre personnage devrait sauter lorsque vous appuyez sur la barre d'espace. L'action `ui_accept` est par défaut connectée à la barre d'espace. Si pour une raison ou une autre ce n'est pas le cas chez vous, il faudra ajuster cela dans les paramètres du projet sous l'onglet CONTRÔLES.

Figure 10.9 : *Notre personnage saute*



Nous avons donc vu comment créer une plateforme, comment marcher dessus grâce à la gravité et comment sauter pour lutter momentanément contre cette gravité !

Vous pourriez créer des niveaux avec ce principe et dupliquer votre plateforme, en créer d'autres et les assembler. Cependant, cette façon de procéder n'est pas optimale. Nous allons voir, dans le prochain chapitre, comment mettre en place un processus permettant de concevoir des niveaux de façon beaucoup plus ergonomique.

11

Mise en place d'un TileSet

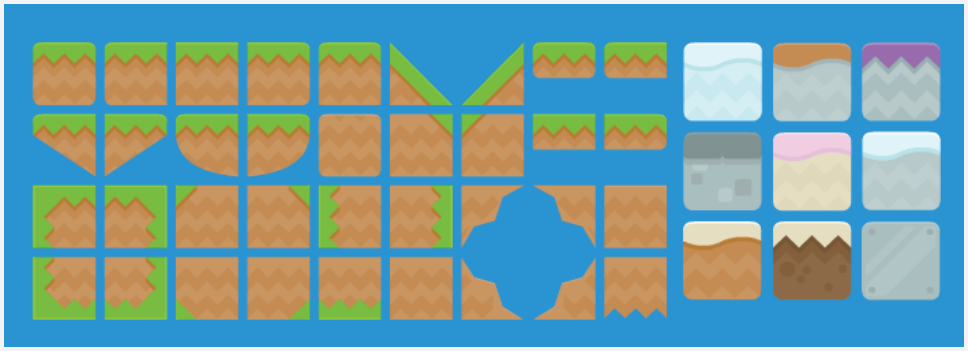
Dans le chapitre précédent nous avons vu comment créer une plateforme sur laquelle notre personnage peut se déplacer. Cela fonctionne bien mais nous n'allons pas créer nos niveaux de cette façon. En effet, cela demanderait beaucoup de temps de créer plusieurs plateformes pour diversifier les niveaux et de les placer une à une. En tant que créateur de jeux, nous ne souhaitons pas nous lancer dans des manipulations fastidieuses. Créer manuellement quelques plateformes pour des besoins spécifiques, pourquoi pas, mais créer 200 niveaux de cette façon, ce n'est pas tenable.

Godot propose un outil spécifique pour la création de jeux 2D. Il s'agit du TileSet. Le TileSet nous permettra d'être beaucoup plus productifs et nous facilitera grandement le travail de création de niveaux.

TILESET ET TILEMAP

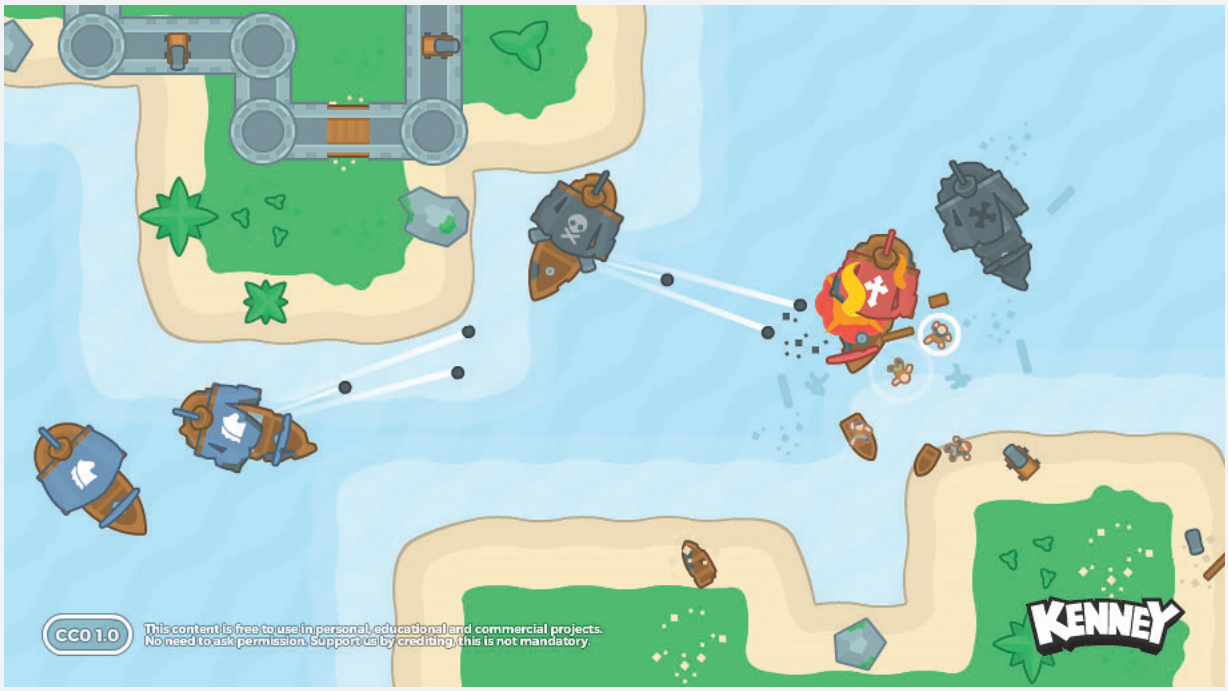
Un TileSet (set de tuiles) est une planche sur laquelle nous retrouvons des tuiles, c'est-à-dire des carrés de 16, 32, 64 (ou plus) pixels de côtés. Ces tuiles découpées permettent de créer les décors des jeux lorsqu'elles sont assemblées. Voici un exemple de TileSet :

Figure 11.1 : *Exemple de TileSet*



Attention, ne confondez pas *TileSet* et *TileMap*. Lorsqu'on parle de TileMap, il s'agit de la *map* (carte/niveau) créée à partir d'un TileSet. Sur la Figure 11.2, nous pouvons voir un niveau (une map) créée à partir d'un TileSet.

Figure 11.2 : *Une map créée à partir d'un set*



Souvent, dans les jeux 2D, les niveaux sont stockés dans des fichiers. Il s'agit des données de votre TileMap. Ces fichiers texte contiennent des chiffres correspondant à des tuiles (tiles). Par exemple 0=vide, 1=mur, 2=eau, 3=herbe, etc. Votre fichier ressemblera à quelque chose dans le genre :

```
1111111111
1223333221
1023333201
1223333221
1111111111
```

Le fichier est alors chargé par le jeu, lu et retranscrit avec les tuiles.

Pour nous simplifier la tâche, nous allons mettre en place un TileSet que nous utiliserons pour créer nos niveaux.

11.1. Création des tuiles

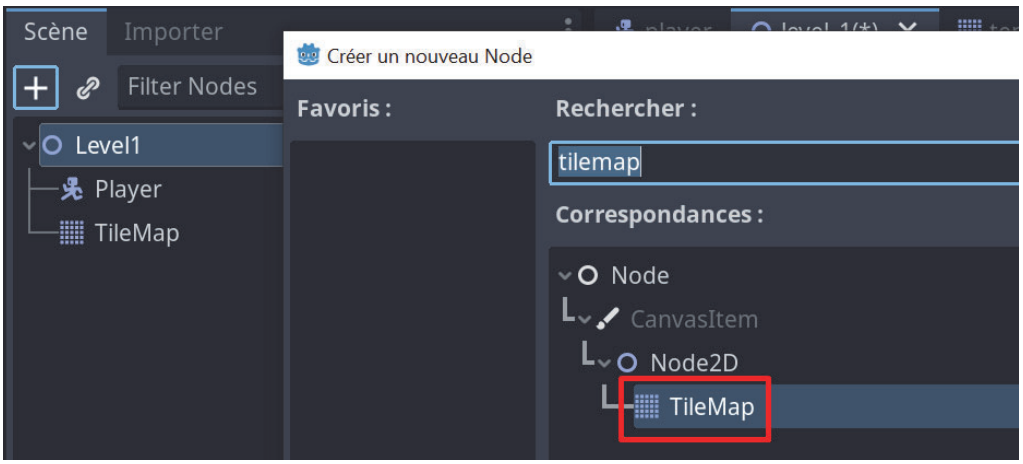
Pour la création de notre TileSet, nous allons utiliser les tuiles que vous avez téléchargées avec les sources du projet. Dans le dossier des assets du jeu, vous retrouverez une image nommée `mini-tileset.png`. C'est cette image d'exemple que nous utiliserons.

Figure 11.3 : *Les tuiles que nous allons utiliser*

Note › *Lorsque vous maîtriserez l'outil de tilemap de Godot, vous pourrez vous rendre sur le site de Kenney et télécharger des tilesets complets (pour mon exemple j'ai uniquement laissé quatre tuiles mais dans un vrai projet on peut avoir des centaines de tuiles). Vous pourrez alors utiliser les techniques que je vais vous présenter à grande échelle. Pour notre projet actuel, j'ai simplifié au maximum.*

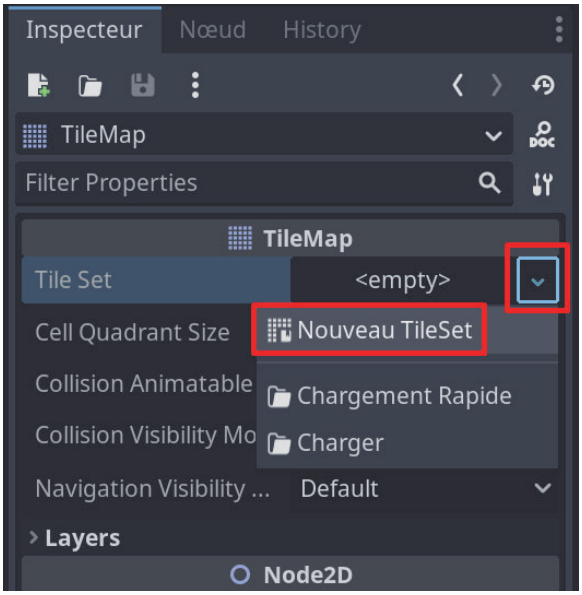
Nous allons travailler sur notre scène `Level1`. Commencez par supprimer les différentes plateformes afin de ne conserver que le nœud principal et le Player. Sélectionnez ensuite le nœud principal et ajoutez-lui un nœud enfant en cliquant sur le bouton PLUS. Choisissez un nœud de type `TileMap`.

Figure 11.4 : *Création du nœud TileMap*



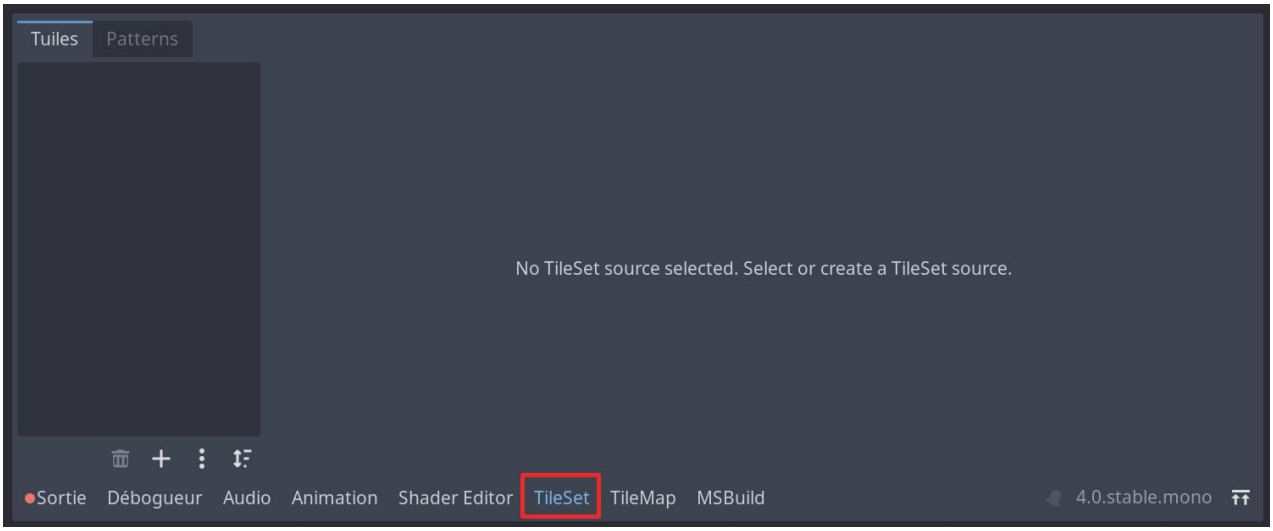
Avec votre TileMap sélectionnée, rendez-vous côté inspecteur. Ici, créez un nouveau TileSet à l'aide du menu déroulant qui s'affiche lorsque vous cliquez sur la flèche.

Figure 11.5 : *Création du TileSet*



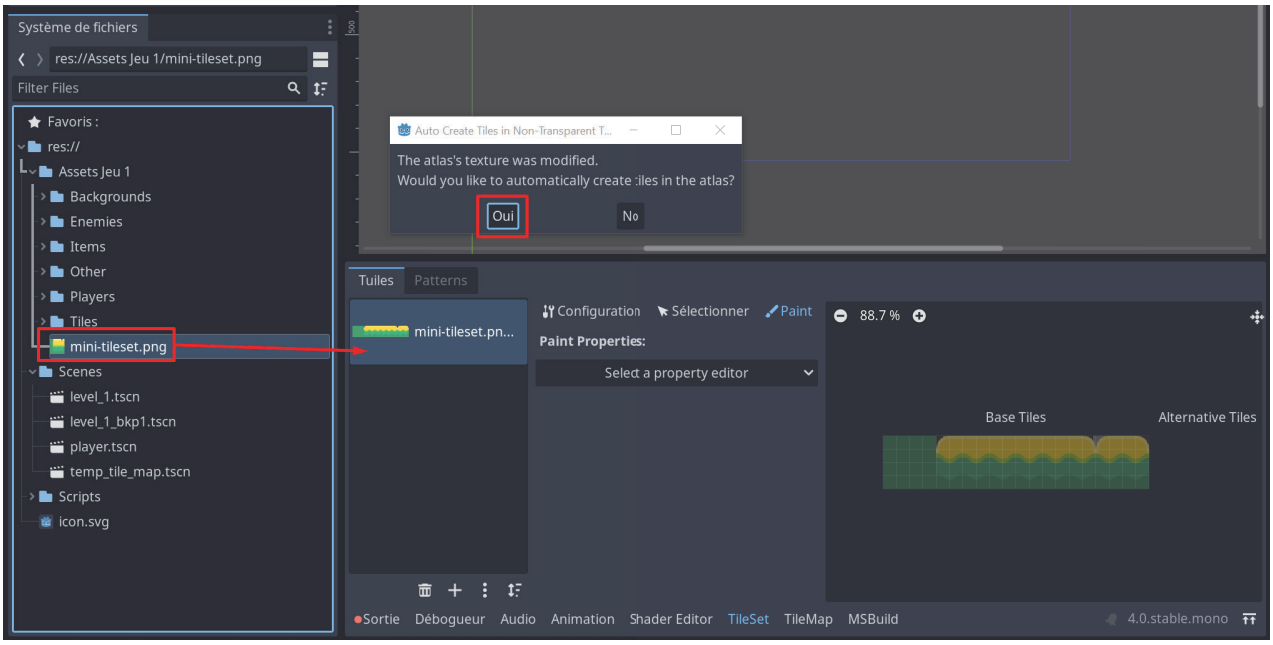
Une nouvelle fenêtre s'ouvre alors en bas de l'interface. C'est là que vous pouvez gérer vos tuiles, les configurer et les utiliser. Pour le moment, vous n'en avez pas. Cliquez sur l'onglet TILESET.

Figure 11.6 : *Affichage du TileSet*



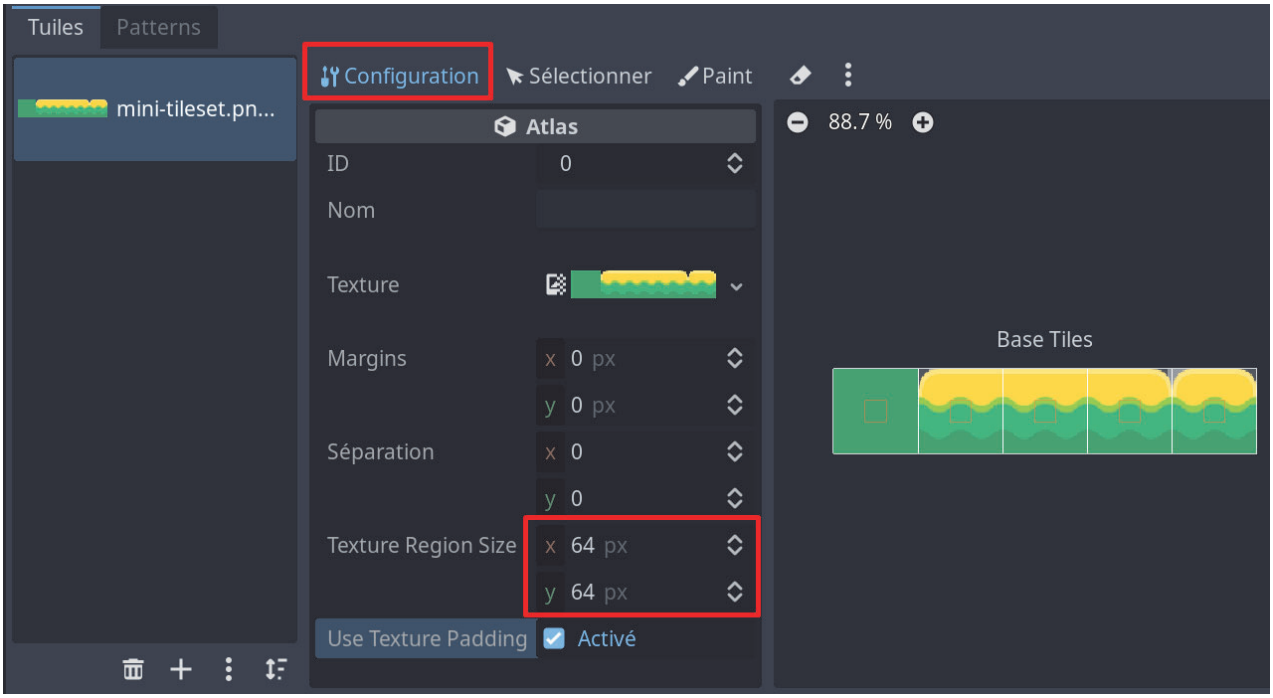
Vous remarquerez une zone plus sombre sur la gauche de la fenêtre. Il s'agit de la zone pour vos tuiles. Faites-y glisser/déposer le mini tileset que j'ai préparé pour vous et qui se trouve à la racine du dossier `Assets jeu 1`. Sur la popup qui apparaît, cliquez sur OUI pour créer les tuiles.

Figure 11.7 : *Générer les tuiles*



Par défaut, les tuiles sont découpées en 16 pixels par 16 pixels. Cependant, les tuiles que nous utilisons font 64×64. Cliquez sur l'onglet CONFIGURATION et définissez une valeur de 64 pour vos textures. De cette façon, la découpe de nos tuiles sera de la bonne taille.

Figure 11.8 : *Configuration de la taille des tuiles*



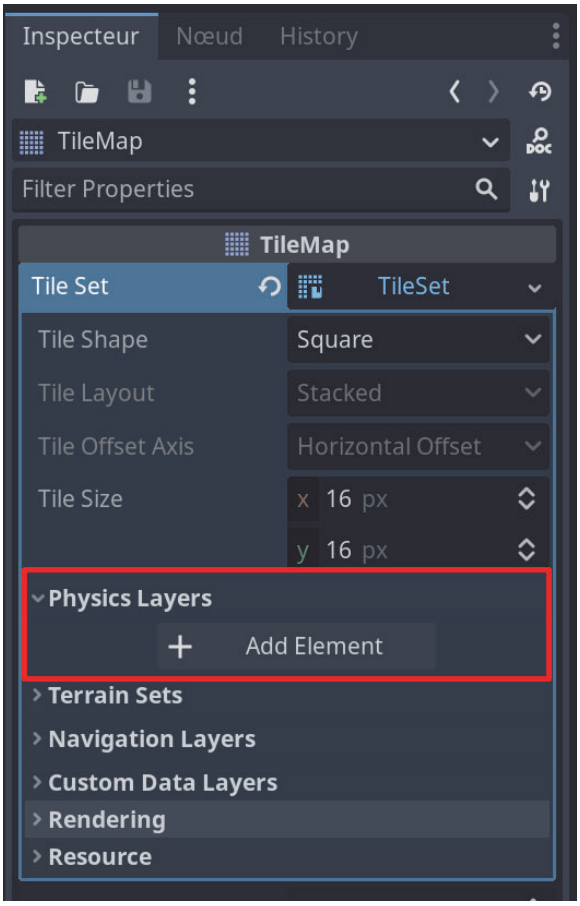
Nos tuiles ne sont pas encore prêtes, il leur manque une propriété de collision. Seul le visuel est configuré. Pour le moment, si le personnage tente de marcher sur ces tuiles, il passera au travers.

11.2. Solidité des tuiles

Notre TileSet va permettre de créer un niveau de jeu. L'assemblage des tuiles formera les plateformes. Par défaut, les plateformes ne sont pas solides. Or nous voulons que le personnage puisse marcher sur les plateformes sans passer au travers. Pour cela il faut ajouter une propriété de collision aux tuiles.

Pour configurer la collision, retournez sur l'inspecteur et cliquez sur le TileSet créé précédemment afin d'afficher ses propriétés. Parmi les propriétés, dépliez PHYSICS LAYER et cliquez sur ADD ELEMENT. Cette manipulation fera apparaître les calques de collision, représentés par deux séries de cases numérotées (voir Figure 11.11).

Figure 11.9 : *Activation de la physique*



Il est possible, lors d'une collision, de tester l'objet touché afin de connaître son type et donc de déclencher une action adaptée.

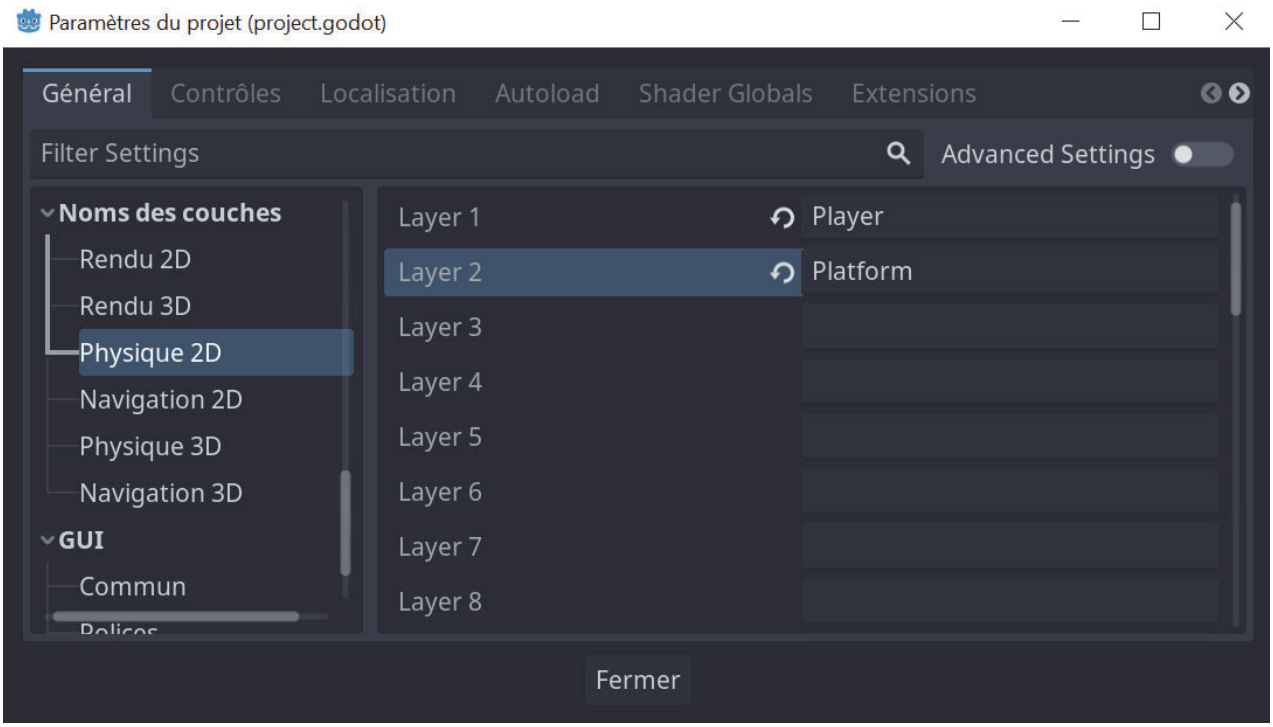
Le système de calques que nous allons mettre en place permet d'indiquer quel objet peut interagir avec quel autre objet. Cela permet de ne tester que les collisions qui nous intéressent et donc d'économiser des tests. Par exemple, le joueur et les plateformes peuvent interagir car le joueur marche sur une plateforme, mais une plateforme ne peut pas interagir avec une autre plateforme. Dans ce deuxième cas, nous ne faisons pas de test et nous économisons de la puissance de calcul.

Les calques (COLLISION LAYER) permettent de classer les objets dans des groupes. Les masques (COLLISION MASK) permettent d'indiquer avec quels autres calques (quels autres groupes d'objets) ils peuvent interagir.

Chaque case numérotée correspond à un calque. Pour faciliter le repérage des calques, vous pouvez leur attribuer un nom via les paramètres du projet. Cliquez donc sur PROJET / PARAMÈTRES DU PROJET, puis sous la rubrique NOMS DES COUCHES/PHYSIQUE2D de l'onglet GÉNÉRAL, créez les calques `Player` et `Platform` en attribuant un nom aux calques 1 (LAYER 1) et 2 (LAYER 2). Le calque 1 sera donc le calque `Player` et le calque 2 le calque `Platform`, et ils seront associés respectivement aux cases 1 et 2 de la propriété PHYSICS LAYER. Une fois créés dans les paramètres, leur nom apparaîtra au survol des cases numérotées dans la propriété (voir Figure 11.11).

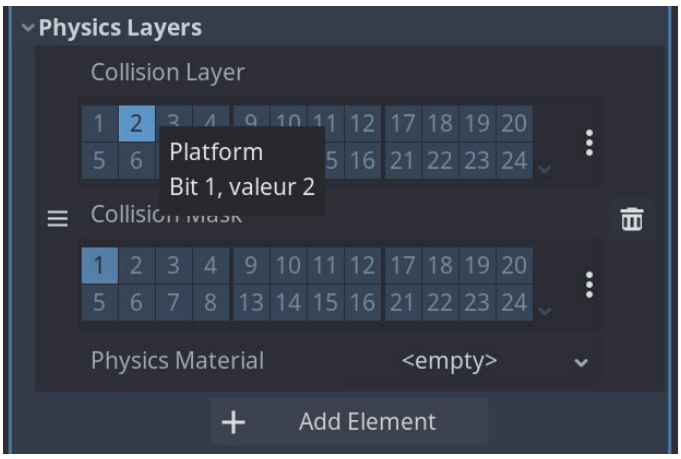
Note *› Le mot anglais layer signifie* couche *en français, mais en infographie le terme de* `calque` *lui est généralement préféré. Sur l'interface en français de Godot, vous trouverez aussi bien couche, calque et layer. Par ailleurs, vous noterez que ce qui est appelé COLLISION MASK désigne également un calque.*

Figure 11.10 : *Création des calques 2D dans les paramètres de projet*



Sélectionnez de nouveau votre Tilemap afin de visualiser les calques. Nous allons cliquer sur la case 2 du bloc COLLISION LAYER et laisser la case 1 cochée du bloc COLLISION MASK. Par ce biais, nous indiquons à Godot que les tuiles sont associées au calque `Platform` et qu'elles peuvent interagir avec les éléments associés au calque `Player`. Le personnage joueur pourra ainsi marcher sur les plateformes conçues avec ces tuiles.

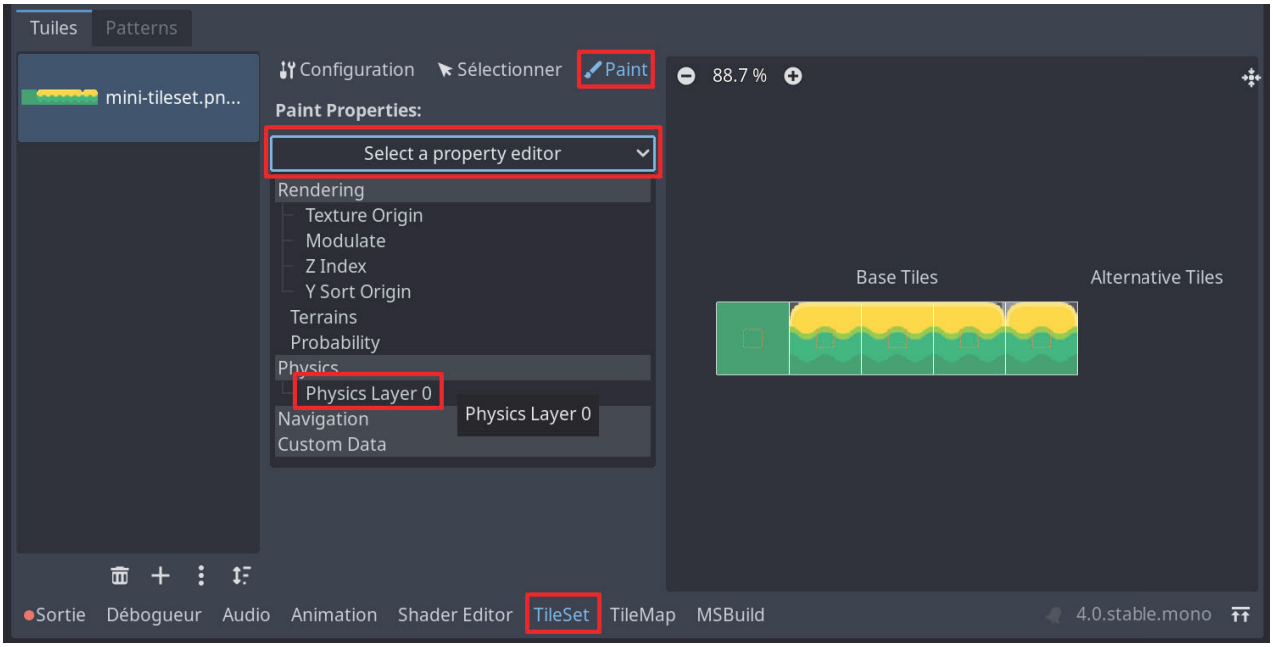
Figure 11.11 : *Sélection des calques*



Note *› Ce système de calques nous sera également utile au chapitre Interaction avec les objets pour détecter si le personnage entre en collision avec les rubis.*

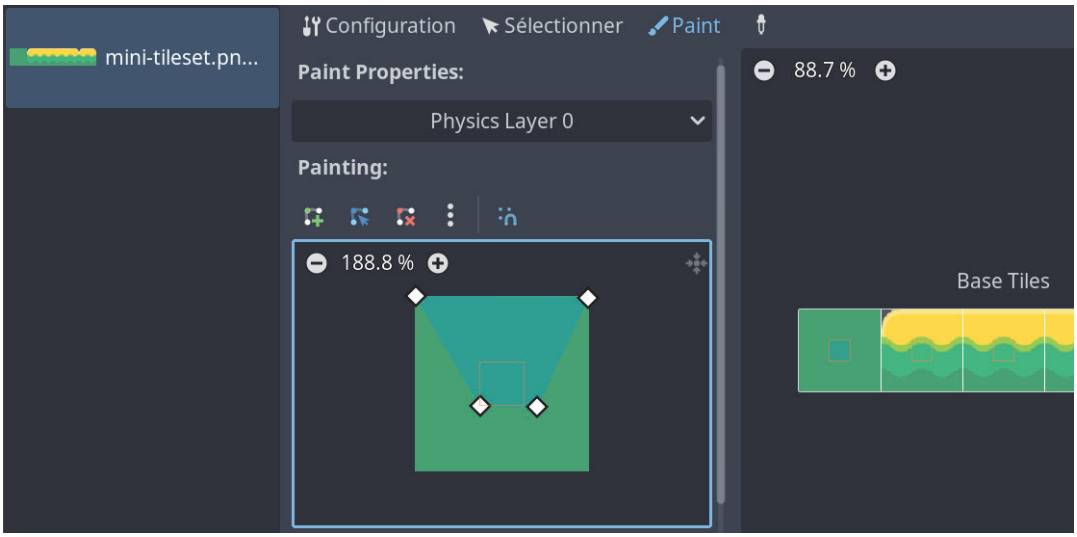
Nous allons maintenant définir la zone de détection des collisions des tuiles (le collider). Retournez dans la fenêtre du bas, onglet TILESET, sous-onglet PAINT et utilisez le menu déroulant pour sélectionner le `Physics Layer 0`.

Figure 11.12 : *Création de la forme du Collider*



Cela fera apparaître une petite zone dans laquelle vous allez pouvoir définir la forme du collider que vous pourrez utiliser pour vos tuiles. Pour visualiser le collider sur une tuile, cliquez sur l'une des tuiles de la zone de droite pour l'afficher sous le collider. Vous pourrez alors modifier la valeur du zoom pour mieux y voir et adapter le collider grâce à ses quatre poignées.

Figure 11.13 : *Visualisation du collider*

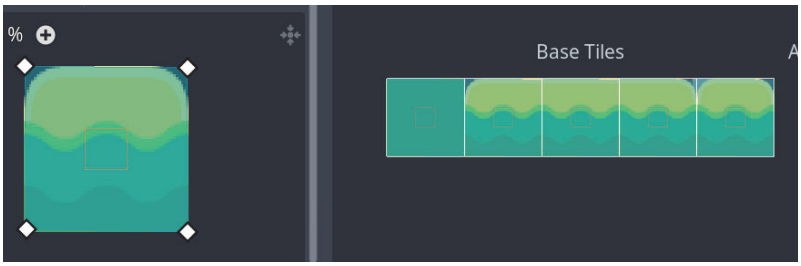


COLLIDER

Un collider est une forme (invisible pour le joueur) qui a pour but de rendre solide un objet de jeu comme par exemple les sprites ou les tuiles. Dans la pratique, il définit la zone de contact. Quand les colliders de deux objets susceptibles d'interagir se touchent alors il y a collision. Nous avons créé le collider de notre personnage en lui ajoutant un CollisionShape2D. Pour les tuiles, nous le faisons via le système de gestion des TileSet. C'est le collider qui rend les tuiles du sol solides.

Votre objectif ici est de modifier la forme du collider afin qu'il soit carré et qu'il recouvre toute la surface de votre tuile. Pas besoin d'une précision extrême, essayez de faire au mieux et ce sera parfait. Ensuite, quand votre collider est bien carré, cliquez sur les différentes tuiles à droite afin d'appliquer et de visualiser le collider :

Figure 11.14 : *Le collider appliqué aux tuiles*



Les tuiles sont prêtes à être utilisées. Nous verrons dans le prochain chapitre comment dessiner notre niveau grâce à ce TileSet.

12

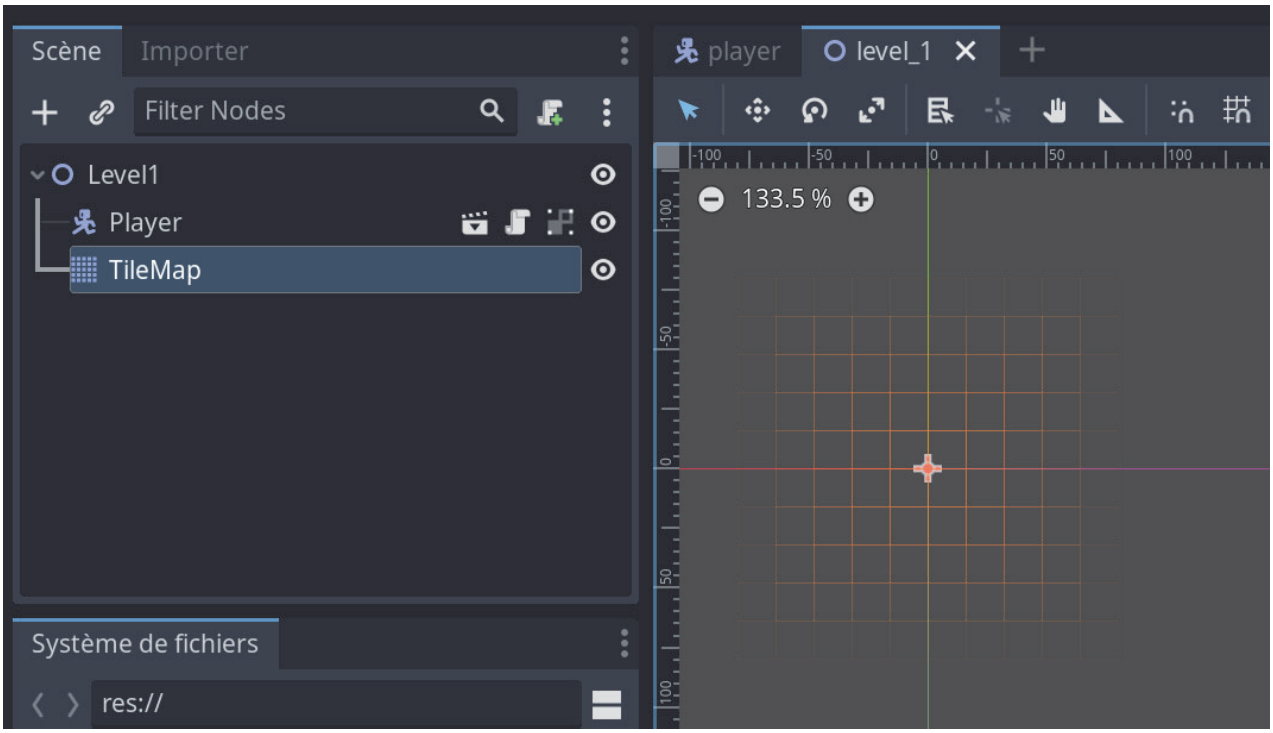
Conception d'un niveau du jeu

Nous allons maintenant créer notre premier niveau à l'aide de notre TileSet.

12.1. Création de la structure de notre niveau

À cette étape, vous devriez être dans la scène `Level1` et n'avoir que le personnage et votre TileMap. Sélectionnez le nœud TileMap afin de laisser apparaître une grille à l'écran. Cette grille vous permettra de mieux visualiser votre niveau pendant que vous placez les tuiles :

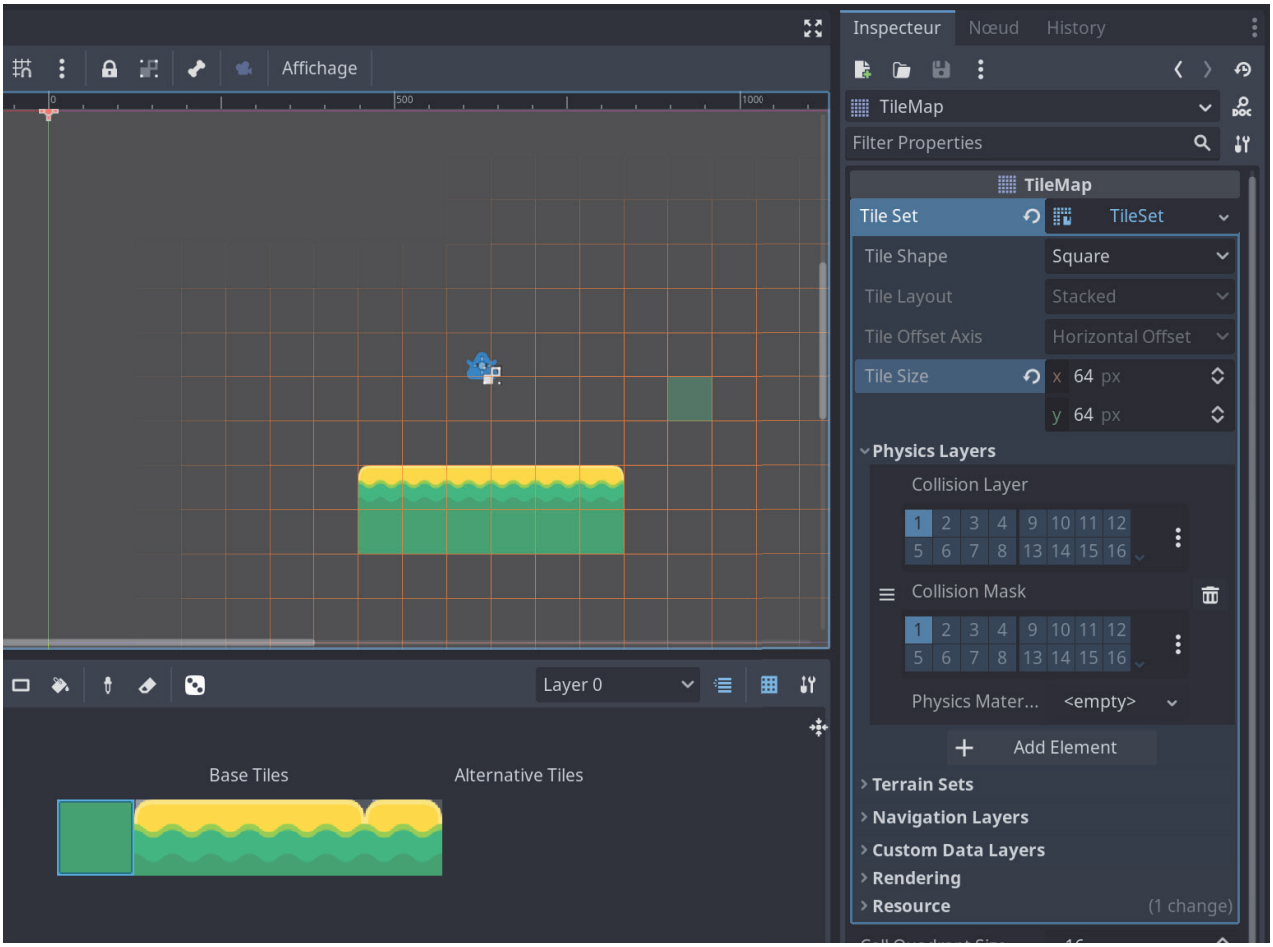
Figure 12.1 : *La grille dans le viewport*



Vous l'aurez compris, cet outil de TileMap vous permet de choisir la tuile voulue en cliquant dessus puis de la placer sur la grille en cliquant à l'endroit souhaité. Vous allez donc sélectionner la tuile à placer puis cliquer dans le viewport afin de créer cette tuile

dans votre niveau. Pensez à changer de tuile quand cela est nécessaire, par exemple pour créer les bordures de vos plateformes.

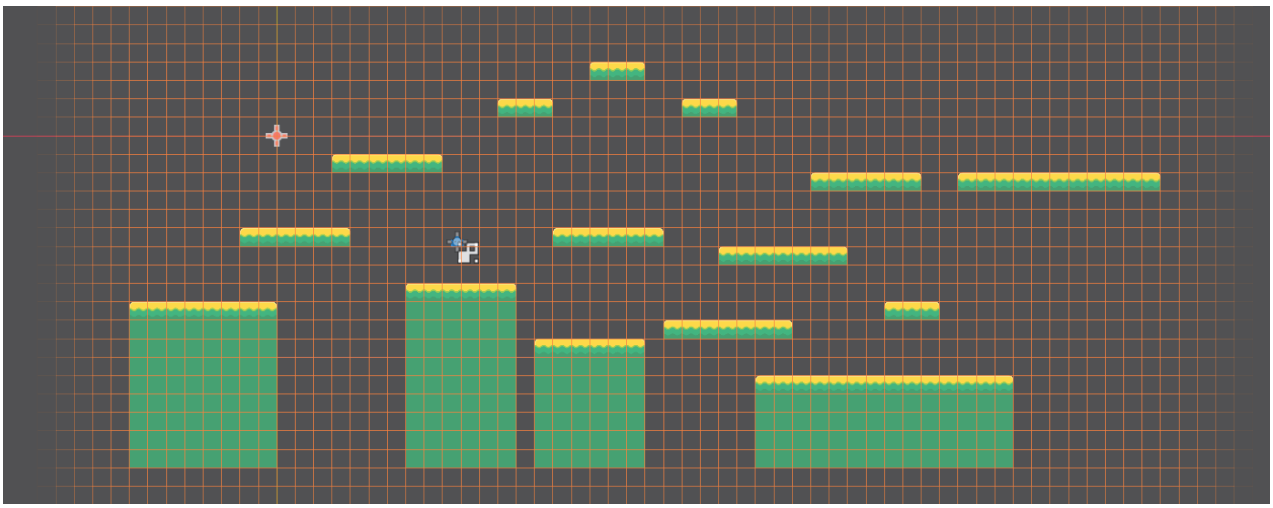
Figure 12.2 : *Utilisation des tuiles*



Note › *Sur la Figure 12.2, vous remarquerez que j'ai modifié dans l'inspecteur la valeur TILE SIZE du TileMap afin que la grille m'affiche des cases de 64px par 64px. De cette façon, nous pouvons peindre nos tuiles dans de bonnes conditions. Vous devez faire cela également de votre côté pour que la taille de votre grille corresponde à la taille de vos tuiles.*

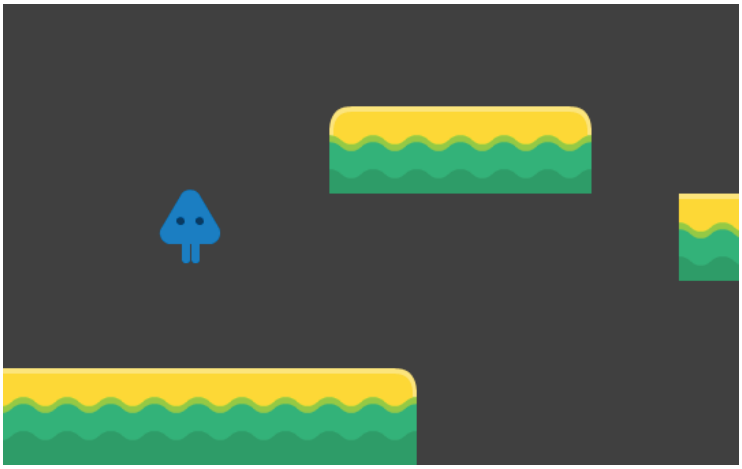
De cette façon, vous pourrez créer votre niveau. Placez des tuiles pour construire des plateformes. Créez le sol, les plateformes ; pensez à utiliser les bords arrondis sur les coins des plateformes pour les rendre plus jolies. Vous pouvez dépasser du carré et sortir du viewport pour créer un grand niveau comme celui de la Figure 12.3.

Figure 12.3 : Exemple de niveau



N'hésitez pas à construire le niveau à votre gré. Placez les tuiles comme bon vous semble. Si vous vous trompez, faites un clic droit pour effacer une tuile. Si vous maintenez enfoncé le bouton de la souris tout en déplaçant celle-ci, vous pouvez peindre plusieurs tuiles à la fois. Puis lancez votre jeu pour le tester !

Figure 12.4 : Notre jeu en action

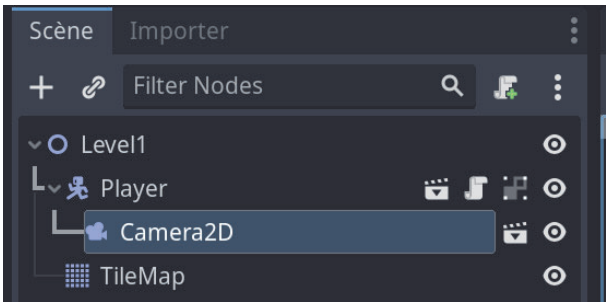


12.2. Mise en place de la caméra

Si vous avez testé votre jeu, vous avez vu qu'il fonctionne bien et que votre personnage peut marcher sur les plateformes créées à partir de tuiles. Cependant, vous constatez un problème. Lorsque vous essayez de sortir de l'écran pour explorer votre niveau, la caméra ne suit pas le personnage. Cela est problématique car vous ne pouvez pas aller en dehors du viewport.

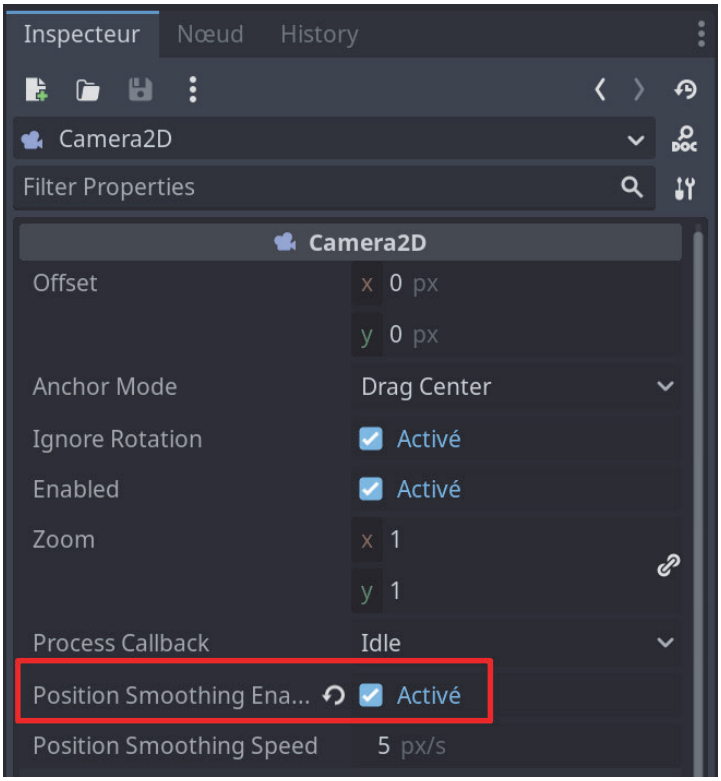
Sauvegardez votre niveau puis créez une nouvelle scène. Dans cette nouvelle scène vide, ajoutez un nœud de type Camera2D en tant qu'enfant du Player. Une fois ceci fait, vous pourrez enregistrer cette nouvelle scène.

Figure 12.5 : *Ajout de la caméra au jeu*



Retournez dans votre niveau 1 et instanciez la caméra. Veillez à bien la centrer par rapport au viewport en la déplaçant. Faites en sorte que son contour (carré coloré) corresponde à celui du viewport. Ajustez à votre guise. Placez-la en tant qu'enfant de Player et cochez la propriété POSITION SMOOTHING ENABLED pour que le mouvement de la caméra soit adouci.

Figure 12.6 : *Configuration de la caméra*



Notre caméra suivra maintenant le personnage de façon fluide et nous serons en mesure d'explorer la totalité du niveau.

Note › *Si vous sélectionnez la caméra et que vous regardez du côté de l'inspecteur, vous verrez le sous-menu LIMIT. Dans ce sous-menu, vous retrouverez les quatre directions (haut, bas, gauche, droite). Si nécessaire, vous pourrez modifier les limites afin de bloquer la caméra sur certains axes. Par exemple, une limite de 0 sur la propriété LEFT empêchera la caméra d'aller plus à gauche.*

12.3. Création d'un arrière-plan en parallaxe

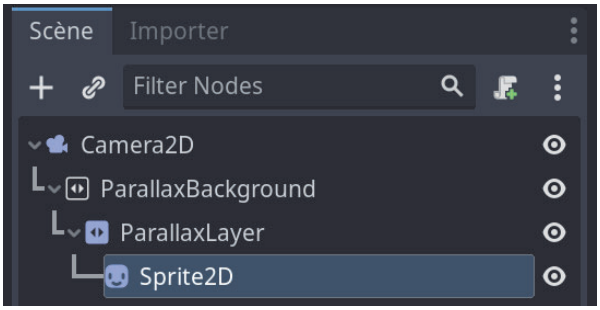
Notre niveau, bien que fonctionnel, est extrêmement simpliste. Pour le moment, il n'est constitué que de quelques plateformes élémentaires. Pour l'enrichir et créer un design attractif, nous allons ajouter de la décoration, à commencer par un arrière-plan en parallaxe. La parallaxe est une technique qui consiste à créer plusieurs fonds qui se déplacent plus ou moins vite selon leur distance supposée du joueur afin de créer un effet de profondeur.

Note › *Pour imager cela, un fond en parallaxe se comporte de la façon suivante : nous pouvons imaginer des montagnes presque fixes et une forêt qui se déplace un peu plus vite. Un peu comme lorsqu'on regarde par la fenêtre de notre voiture. Ce qui se trouve près de nous comme les arbres ou les panneaux défilent très vite et les montagnes au fond sont quasiment fixes. C'est cet effet que nous allons reproduire.*

Retournez sur la scène de la caméra afin de créer ce système de parallaxe. Créez un nœud enfant de type ParallaxBackground. Ce ParallaxBackground sera le conteneur qui contiendra l'ensemble des plans (des calques) de notre arrière-plan.

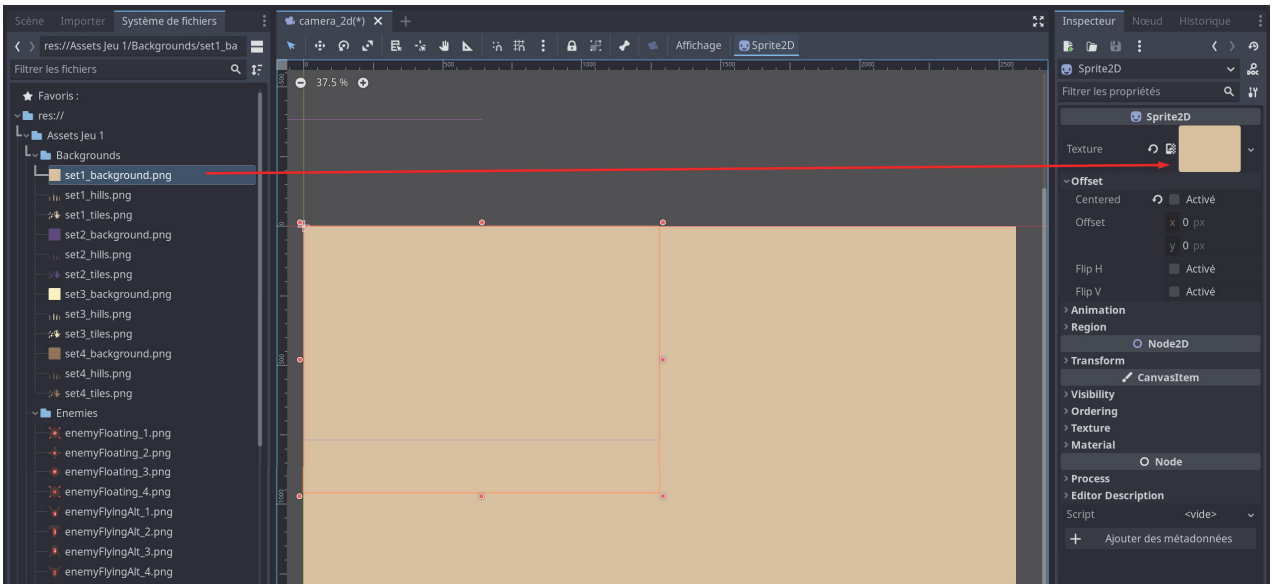
Nous commençons par créer un calque [layer]. Ajoutez un nœud enfant au ParallaxBackground. Votre nœud doit être de type ParallaxLayer. Ce ParallaxLayer doit lui-même avoir un enfant de type Sprite2D qui sera le visuel. Créez ce nœud enfant. Vous devriez avoir une scène comme celle de la Figure 12.7.

Figure 12.7 : *Notre scène*



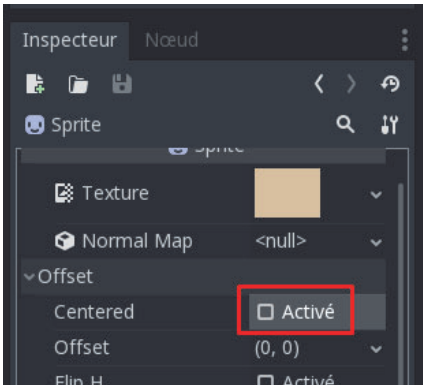
Nous allons ajouter une texture au Sprite de notre premier layer. Ce premier élément sera celui qui sera le plus au fond du décor. Il sera immobile. Parmi les ressources livrées avec ce livre, vous retrouverez des images d'arrière-plan dans le dossier Backgrounds. Choisissez la couleur qui vous plaît. Pour ma part, je sélectionne le fond uni beige.

Figure 12.8 : *Ajout d'un fond uni*



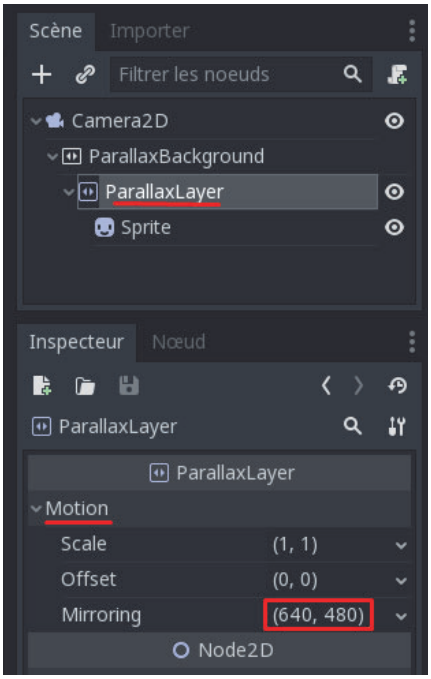
Dans l'inspecteur, sur votre Sprite sous la propriété OFFSET, décochez la case ACTIVÉ du paramètre CENTERED.

Figure 12.9 : *Désactivation d'une option*



Cliquez ensuite sur le ParallaxLayer et regardez dans la propriété MOTION l'option MIRRORING. Mettez une valeur de 640 × 480 qui correspond à la taille de notre image de fond.

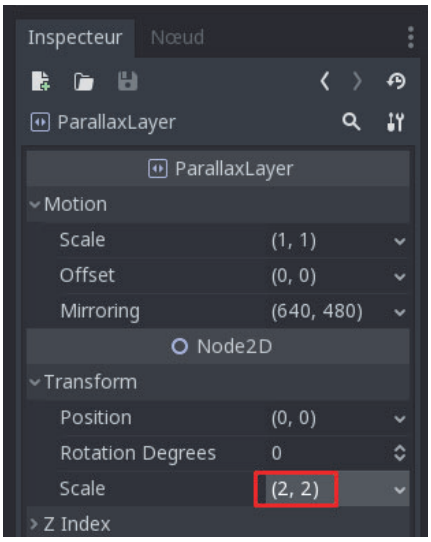
Figure 12.10 : *Activation du mirroring*



Toutes ces options que nous venons de mettre en place permettent de créer un système qui répète l'arrière-plan à l'infini durant le déplacement du personnage. De cette façon, le joueur verra toujours un décor d'arrière-plan même s'il se déplace.

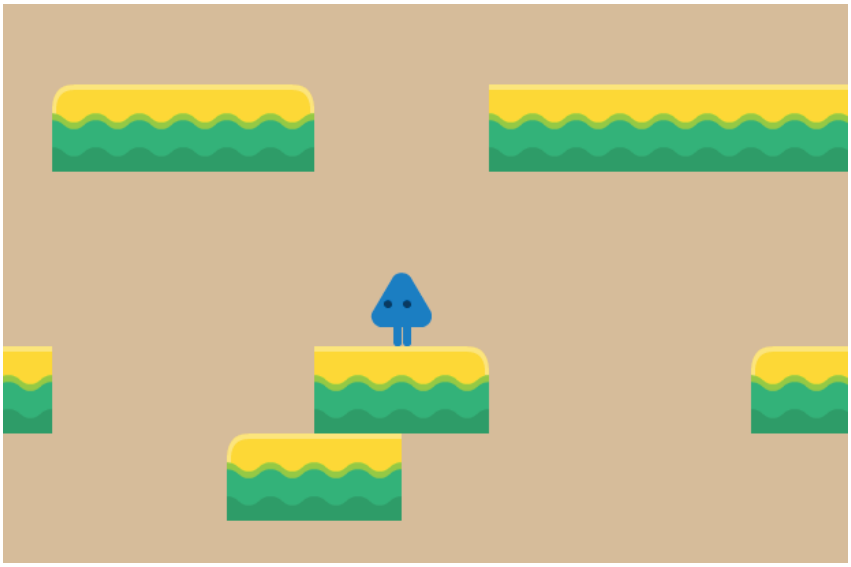
Actuellement, étant donné la résolution de notre fenêtre, il se peut que certains endroits ne soient pas couverts par l'image de fond. Nous allons agrandir la taille de notre arrière-plan afin d'être sûr que tout l'écran soit bien couvert. Cliquez sur le ParallaxLayer et, dans l'inspecteur, sous la rubrique TRANSFORM, mettez le SCALE à `0,0`.

Figure 12.11 : *Modification du Scale*



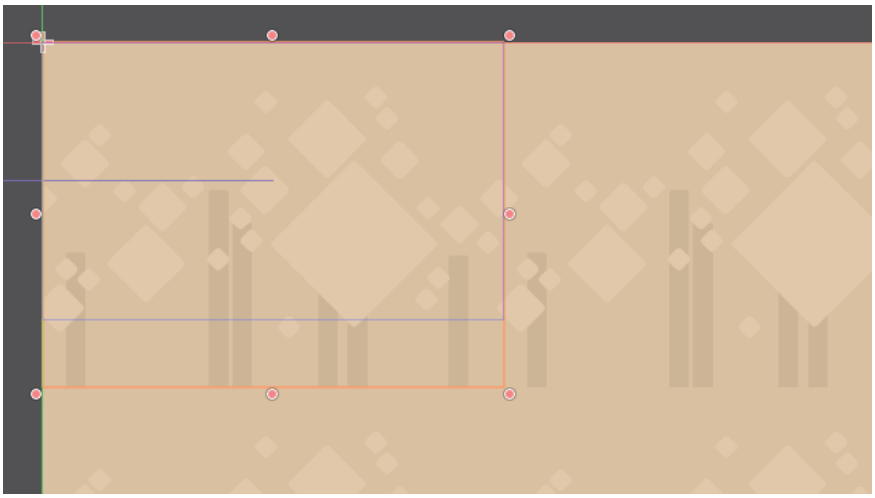
Vous pouvez retourner sur la scène de votre jeu après avoir sauvegardé vos modifications et vérifier que tout fonctionne. Voici ce que vous devriez avoir :

Figure 12.12 : *Notre couleur de fond*



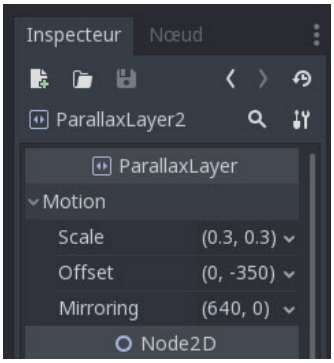
Retournez sur la scène de la caméra où nous avons mis en place l'arrière-plan en parallaxe. Dupliquez deux fois le ParallaxLayer actuel afin d'avoir trois calques à votre disposition. Modifiez la texture des sprites de vos calques 2 et 3 afin de diversifier votre décor. Pour le layer numéro 2, j'utiliserai les formes rectangulaires, pour le numéro 3, les losanges. Cela nous donnera le résultat de la Figure 12.13.

Figure 12.13 : *Notre arrière-plan à trois images*



Pour que l'arrière-plan soit correctement configuré, nous devons encore ajuster quelques paramètres. Cliquez sur le second ParallaxLayer et mettez, sous la rubrique MOTION, un SCALE de `0.3,0.3`, un OFFSET de `0,-350` et un MIRRORING de `640,0`.

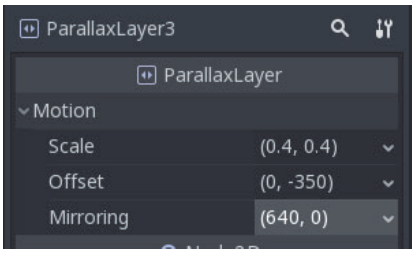
Figure 12.14 : *Configuration du calque 2*



La propriété MOTION correspond au mouvement du calque. Le SCALE indique la quantité de mouvement que doit effectuer l'image par rapport au mouvement du personnage. Notre image se déplacera légèrement à chaque déplacement du personnage. L'OFFSET nous permet de décaler l'image afin de la centrer un peu mieux à l'écran. Enfin, nous mettons une valeur de 0 en Y au niveau du MIRRORING afin de n'avoir une répétition qu'en horizontal et pas en vertical.

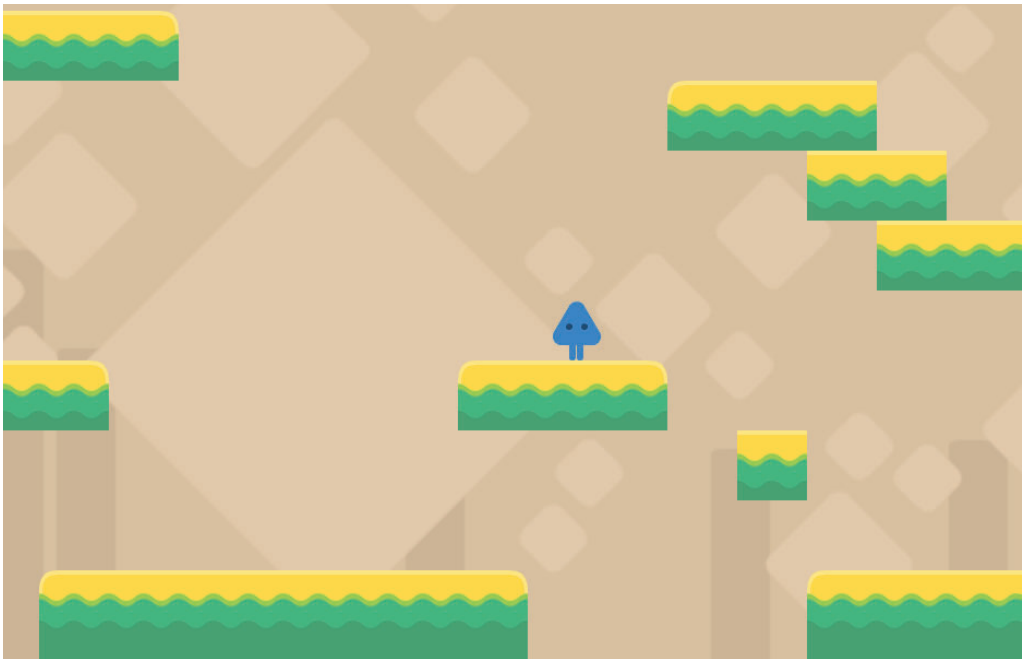
Toutes ces valeurs peuvent être ajustées à votre guise. J'ai choisi les miennes après quelques essais et je les ai conservées car ce sont celles qui me convenaient le mieux mais tout dépendra du fond utilisé. Chaque fond aura des valeurs spécifiques. Occupons-nous du layer numéro 3. Voici les réglages que j'ai utilisés :

Figure 12.15 : *Réglages du layer 3*



Vous remarquerez que c'est sensiblement la même chose que pour le layer 2. La seule différence se trouve dans le SCALE qui est légèrement plus élevé. Cela est dû au fait que ce layer est un peu plus proche du joueur et doit donc se déplacer un peu plus rapidement. Vous pouvez maintenant tester votre jeu et vérifier que votre parallaxe fonctionne correctement.

Figure 12.16 : *Parallaxe*



Vous pouvez maintenant voir sur cette image les différents plans de notre arrière-plan. Pour peaufiner votre décor, glissez/déposez depuis les assets quelques éléments visuels décoratifs comme des plantes, des panneaux ou des fleurs. Vous pouvez simplement placer les éléments, les importer en tant que Sprite et tout regrouper dans un nœud simple afin d'organiser votre niveau.

Figure 12.17 : *Quelques objets décoratifs*

Voilà pour ce chapitre. Avant de passer à la suite, essayez de bien décorer votre niveau afin de le rendre le plus plaisant visuellement possible.

13

Interaction avec les objets

La plupart des jeux fonctionnent avec un système d'interaction entre le personnage et les objets du jeu. Je pense par exemple aux objets à ramasser, aux pièces ou cristaux à collectionner, aux clés à récupérer pour ouvrir des portes, etc. Ces interactions permettent de mettre en place des énigmes ou des challenges qui vont rendre votre jeu plus intéressant.

Pour comprendre comment mettre en place un tel système, nous allons créer des pierres précieuses à ramasser et nous les placerons un peu partout dans notre niveau. Ces pierres pourront être utilisées comme de l'argent pour acheter des objets ou comme des bonus donnant des vies supplémentaires lorsque nous aurons implémenté un système de vies.

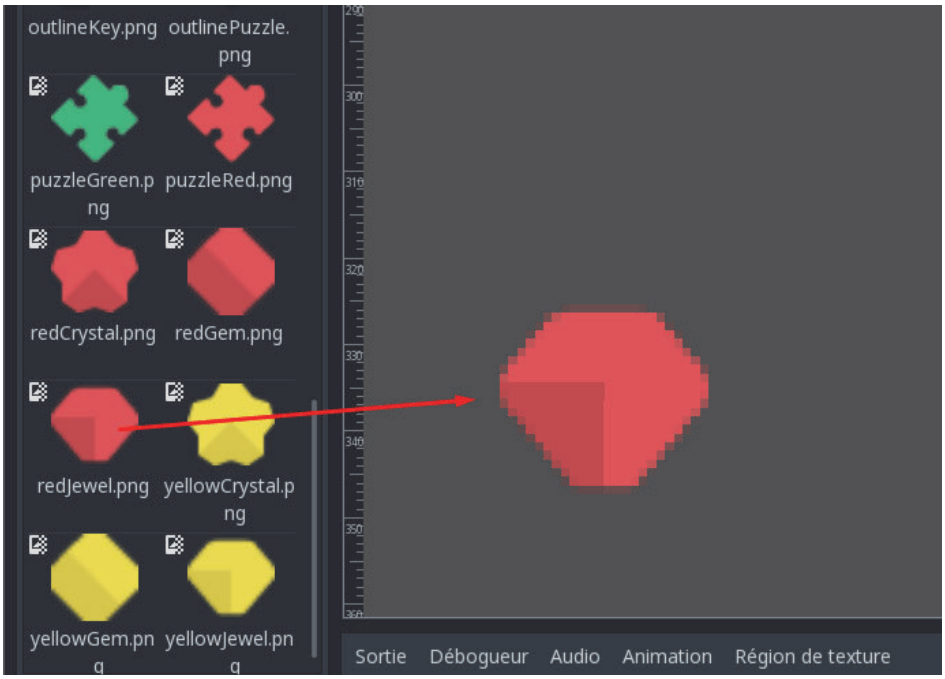
Nous allons commencer par créer ces pierres, les configurer puis créer le script permettant au personnage de les ramasser lors d'une collision entre celui-ci et la pierre précieuse.

13.1. Création d'une pierre précieuse

Dans tous les jeux Zelda (Nintendo), Link, le héros, peut ramasser des rubis. Ces rubis sont utilisés comme monnaie dans Hyrule, le monde où se déroule l'aventure. Un rubis vert vaut 1, un bleu vaut 5, un rouge vaut 20, un violet 50, etc. Nous allons mettre en place le même principe. Nous ne nous concentrerons pour notre exemple que sur un équivalent du rubis vert.

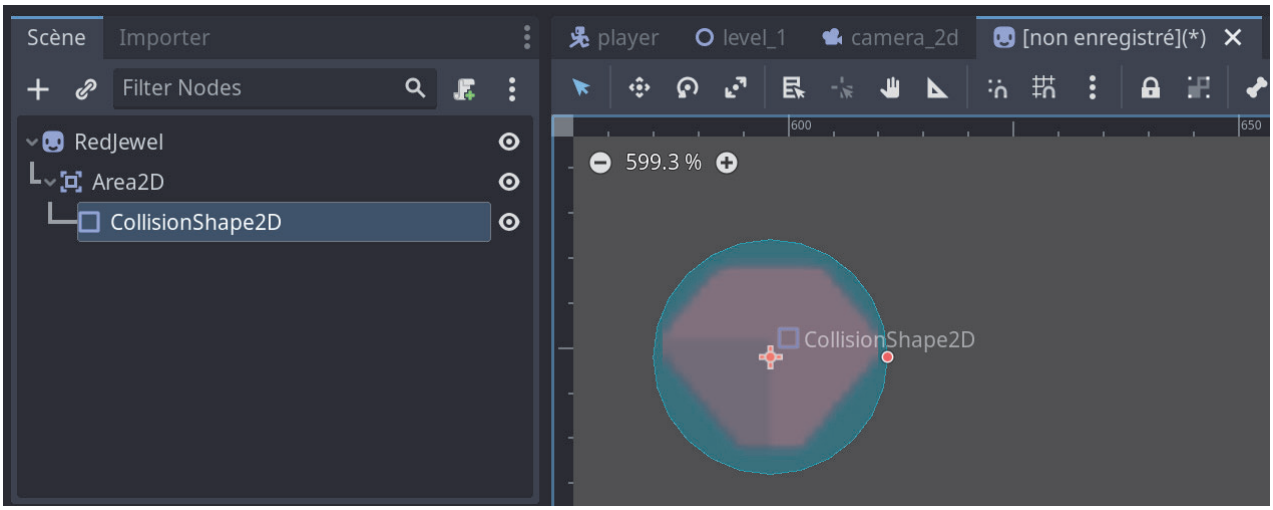
Créez une nouvelle scène dans laquelle nous allons créer une pierre précieuse. Parmi les assets, vous trouverez dans le dossier `Items` différents bijoux. J'utiliserai pour ma part la pierre rouge pour avoir un meilleur contraste avec le reste des éléments de mon niveau. Faites glisser cette pierre sur votre scène.

Figure 13.1 : *Ajout d'une pierre précieuse*



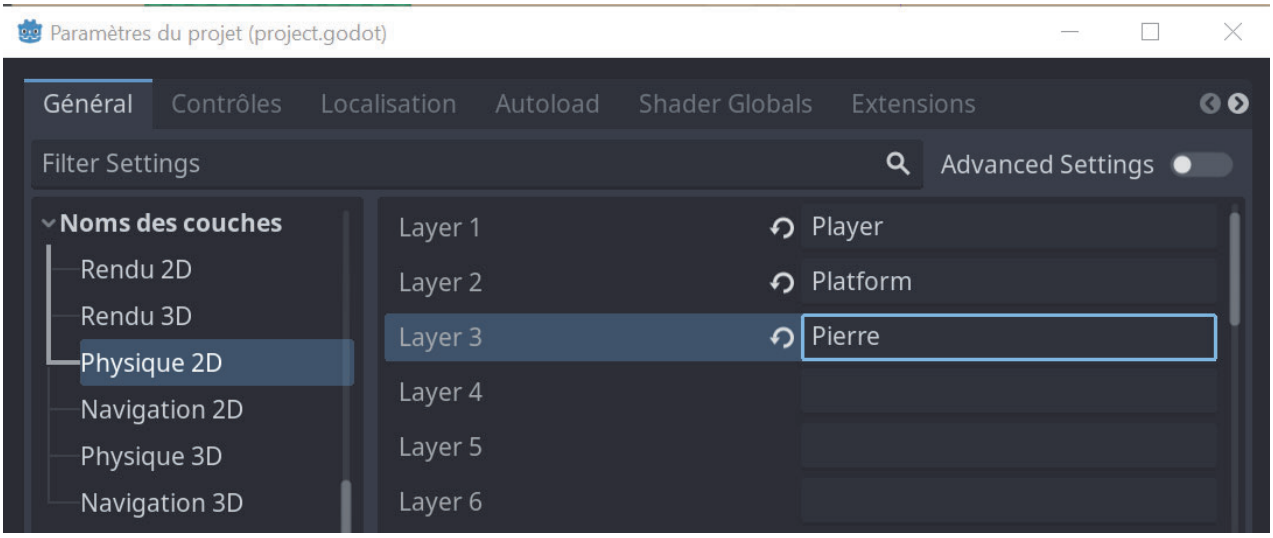
Pour pouvoir gérer les collisions, nous allons ajouter un enfant à notre Sprite qui sera un Area2D. Ce composant permet de tester des collisions mais il a besoin d'un CollisionShape pour cela. Ajoutez donc ce CollisionShape2D et créez un Shape de type CircleShape. Pensez à bien ajuster le composant de collision.

Figure 13.2 : *Mise en place de la gestion de collision*



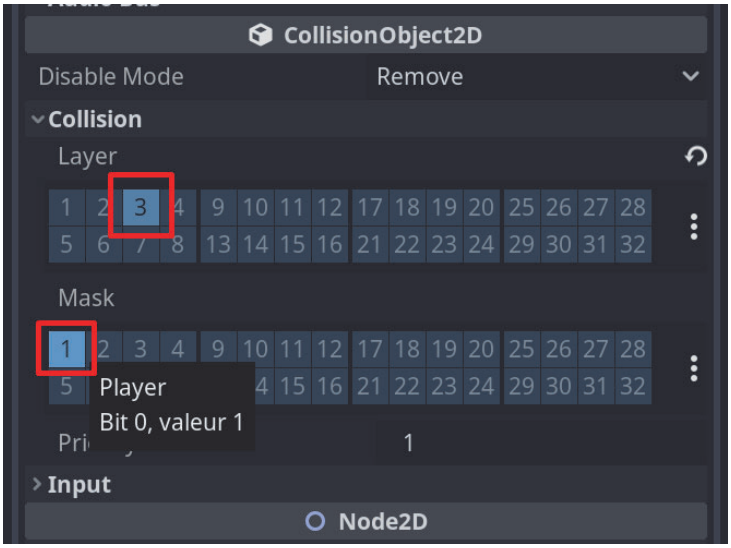
Nous devons maintenant placer notre pierre dans un calque précis et indiquer le masque avec lequel elle peut interagir (voir Section 11.2, Solidité des tuiles). Retournez donc dans les options du projet à la rubrique PHYSIQUE 2D et créez un nouveau calque `Pierre`.

Figure 13.3 : *Création du calque*



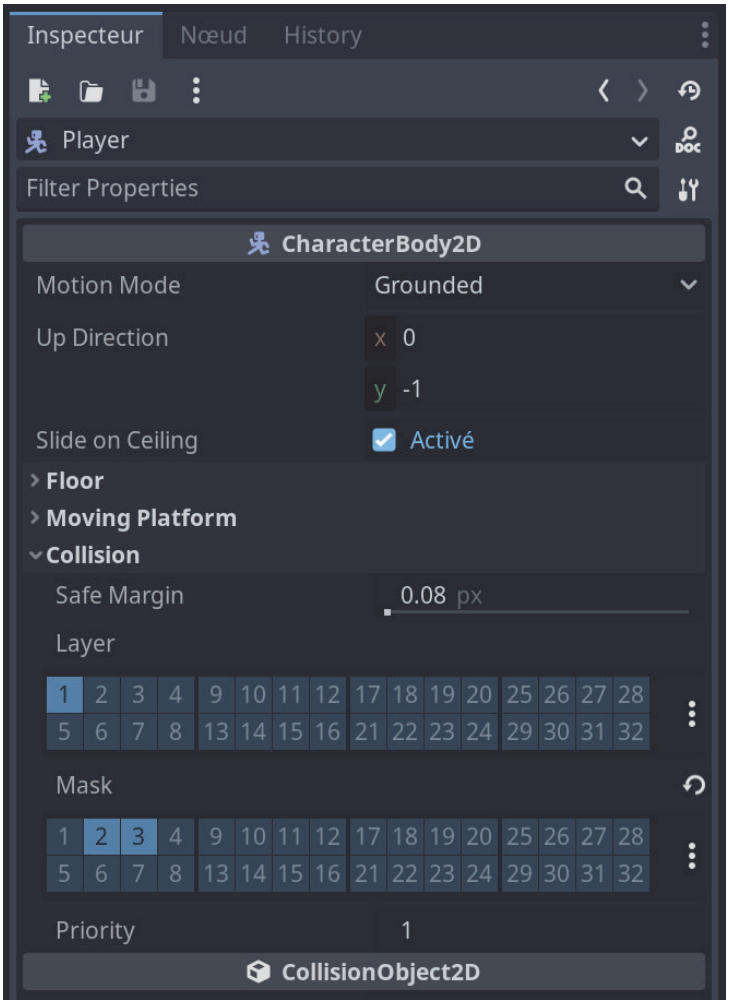
Retournez ensuite sur la pierre, sur son Area2D, et placez-la sur son calque en cochant la case 3 puis faites-la interagir avec le masque `Player` en cochant la casee 1.

Figure 13.4 : *Configuration du calque*



Pensez également à retourner sur la scène de votre Player afin de modifier sa collision pour qu'il puisse interagir aussi avec le masque `Pierre`. Veillez à toujours bien enregistrer vos modifications pour éviter les mauvaises surprises.

Figure 13.5 : *Propriétés du Player*



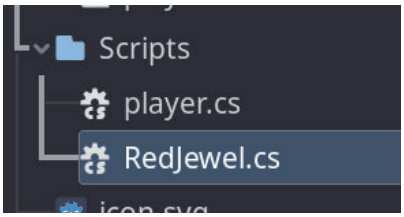
Notre personnage et notre pierre sont configurés comme il faut ! Nous pouvons maintenant programmer les interactions.

13.2. Création du script de ramassage

Penchons-nous maintenant sur l'écriture du script qui permettra au personnage de ramasser les pierres rouges lorsque celui-ci les touchera. Ce script pourra parfaitement être adapté au ramassage de pièces ou de n'importe quel autre type d'objets. Vous pourrez l'ajuster selon vos besoins.

Retournez sur la scène de la pierre précieuse et cliquez sur celle-ci dans la fenêtre Scène. Cliquez ensuite sur l'icône de création de scripts. Donnez un nom à votre script et enregistrez-le dans le dossier des scripts.

Figure 13.6 : *Création du script de ramassage*

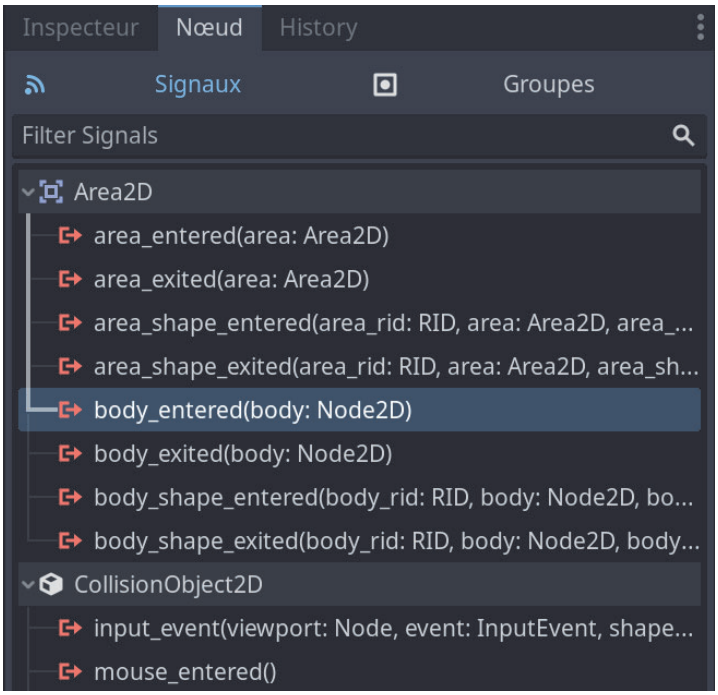


Par défaut, le script contiendra les fonctions `ready` et `process`. Supprimez la fonction `process`, nous n'en n'aurons pas besoin. Nous allons plutôt avoir besoin d'une fonction qui se déclenchera lors d'une collision.

Lorsque vous ajoutez un composant Area2D à un objet (ce qui est le cas de notre pierre précieuse), vous avez accès à un certain nombre de fonctions spécifiques à ce composant, dont une fonction `body_entered`. Cette fonction se déclenche lorsqu'un Body entre dans l'Area2D. Pour retrouver cette fonction, sortez de votre script pour retourner sur la scène de votre pierre précieuse et cliquez sur son Area2D.

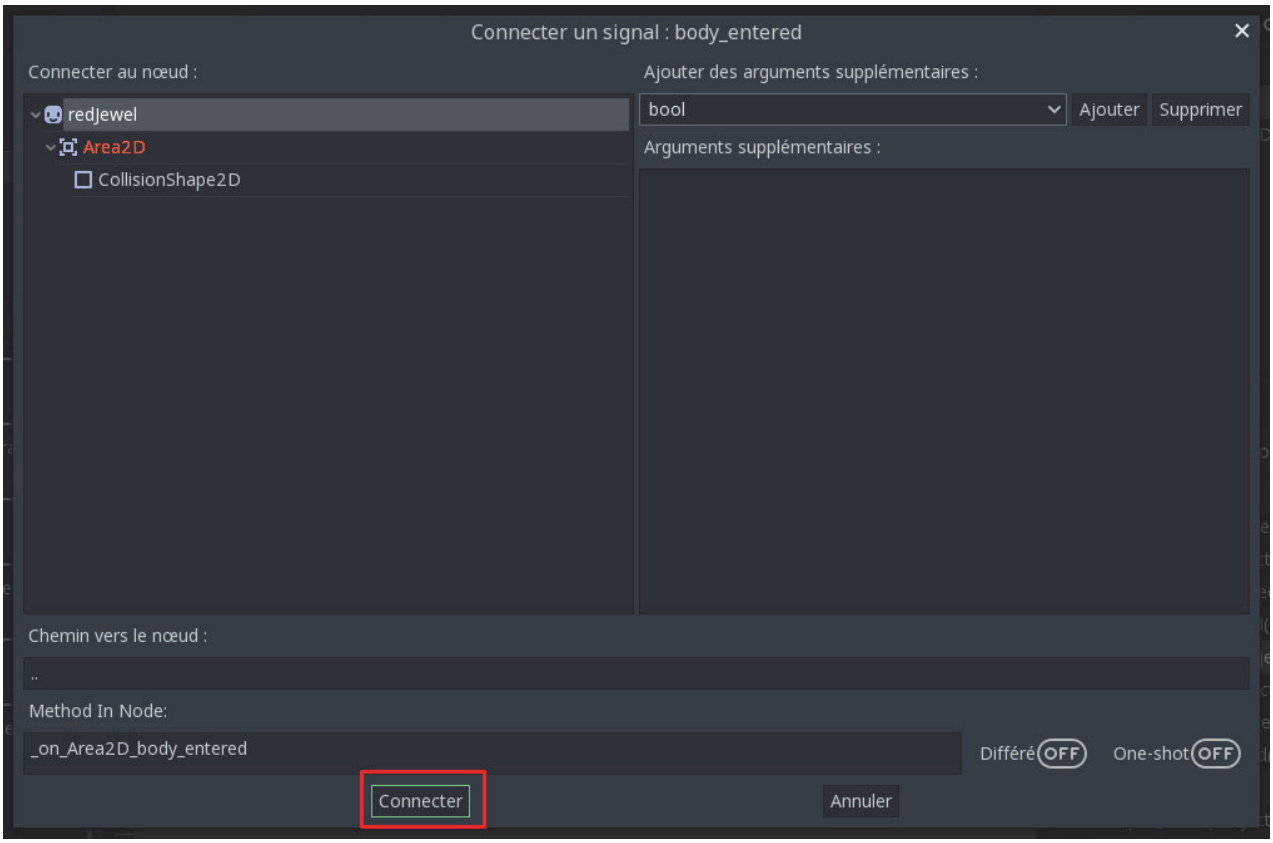
Avec l'area sélectionnée, cliquez sur l'onglet NŒUD dans lequel sont regroupées toutes les fonctions propres au Area2D, et en particulier la fonction `body_entered`.

Figure 13.7 : *Liste des fonctions disponibles*



Cliquez sur cette fonction puis cliquez sur CONNECTER en bas de la fenêtre NŒUD. Une nouvelle fenêtre s'ouvre et vous demande de choisir l'objet qui sera connecté. Laissez tout par défaut et cliquez sur CONNECTER.

Figure 13.8 : *Connecter la fonction*

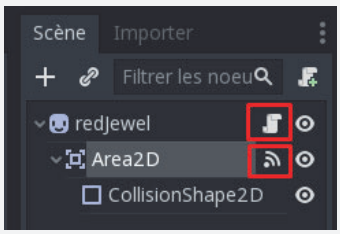


Cette action créera une nouvelle fonction `_on_Area2D_body_entered` dans votre script.

VOTRE SIGNAL EST-IL BIEN CONNECTÉ AU NŒUD ?

Pour connecter un signal à un nœud, il faut que le nœud en question possède un script et que celui-ci ait été enregistré. Vous pouvez savoir qu'un script est bien attaché grâce à l'icône Script. Vous pouvez savoir également qu'un signal est correctement connecté grâce à son icône.

Figure 13.9 : *Icônes Script et Signal*



Si vous essayez de connecter un signal à un nœud n'ayant pas de script, une erreur surviendra.

Par défaut la fonction est mal positionnée. En effet, elle est à l'extérieur de la classe. Coupez le code pour le coller dans la classe comme cela :

```
using Godot;
using System;

public partial class RedJewel : Sprite2D
{

  public override void _Ready()
  {

  }

  private void _on_area_2d_body_entered(Node2D body)
  {

  }
}
```

À l'intérieur de la fonction `_on_Area2D_body_entered`, nous allons décrire ce qui doit se produire lors de la collision entre le personnage et l'objet. Notre objectif est d'augmenter la richesse du personnage de 1 puis de faire disparaître le bijou car il a été ramassé.

Pour le moment, nous n'avons pas mis en place de système d'argent au niveau de notre personnage. Aucune variable ne stocke la quantité de pierres précieuses que le personnage possède. Nous allons rectifier cela en ajoutant une variable `public int coins = 0;` au script `Player`. Une fois ceci fait, retournez dans le script du bijou.

Note › *J'ai choisi d'appeler ma variable `coins` [pièces] par habitude car en général dans les jeux vidéo le personnage ramasse des pièces d'or. On aurait tout à fait pu choisir un autre nom. Dans le cadre de ce projet, le personnage ramassera des pierres précieuses mais cela restera la monnaie de notre jeu.*

Afin de mettre en place un système de récolte des objets, nous devons être en mesure de communiquer au script du joueur si un objet a été ramassé. Pour cela plusieurs possibilités s'offrent à vous. Vous pouvez passer par un signal ; créer une référence vers le joueur ; utiliser une variable globale ; etc. Dans mon cas, bien que ce ne soit pas la meilleure approche (je vous explique pourquoi un peu plus bas), je vais utiliser une référence et la méthode `GetParent` pour récupérer le nœud du joueur. Cela me permettra de vous présenter l'utilisation de `GetParent` qui vous sera utile dans de nombreuses situations. Je vous proposerai une alternative à cette façon de procéder dans une note à la fin de cette section.

La première chose à faire est de créer une variable `public player p;` qui sera une référence vers le script du personnage car nous aurons besoin d'y accéder pour modifier la valeur de `coins`. Nous devons renseigner cette variable une fois que le jeu est lancé

et que tous les scripts sont prêts. Nous utiliserons pour cela la fonction `ready`. Dans cette fonction nous allons récupérer le script du joueur afin de le stocker dans notre variable.

Pour récupérer un script se trouvant sur un objet de la scène, nous devons récupérer le parent du rubis avec `GetParent()`. Le parent de notre rubis sera le nœud principal de la scène et à partir de là, nous pourrons accéder aux autres nœuds présents dans la scène. Pour récupérer un nœud, nous devons utiliser la fonction `GetNode`. La ligne de code complète permettant de récupérer le script du joueur est la suivante.

```
p = GetParent().GetNode<player>("Player");
```

Cette ligne nous permet de récupérer le script `Player` et de le stocker dans sa variable. Cela nous sera très utile pour accéder rapidement aux variables de Player notamment `coins`, indiquant la quantité de pierres précieuses qu'il possède.

Dans la fonction `_on_Area2D_body_entered`, nous allons pouvoir déclencher l'ajout d'un rubis dans l'inventaire du personnage. Pour cela, écrivez :

```
PlayerScript.coins += 1
```

Si vous souhaitez vérifier que tout fonctionne bien, affichez le nombre de pierres dans la console de cette façon :

```
GD.Print(p.coins);
```

Ainsi, à chaque collision avec un bijou, vous verrez le nombre de rubis dans l'inventaire du personnage.

Pour terminer, comme la pierre a été récupérée, nous devons la faire disparaître. Si nous ne faisons pas cela, elle pourra être récupérée un nombre infini de fois. Pour détruire un objet, nous devons utiliser la fonction `QueueFree()`. Le script complet de votre pierre précieuse doit être le suivant :

```
using Godot;
using System;

public partial class RedJewel : Sprite2D
{
  public player p;

  public override void _Ready()
  {
    p = GetParent().GetNode<player>("Player");
```

```
  }

  private void _on_area_2d_body_entered(Node2D body)
  {
    p.coins ++;
    GD.Print("Nombre de pierres : " + p.coins);
    QueueFree();
  }
}
```

ATTENTION AUX RÉFÉRENCES

J'ai présenté ici une façon de procéder via l'utilisation d'une référence et de la fonction `GetParent`. Comme indiqué plus haut, cette façon de faire peut poser un problème dans certains cas (par exemple si votre arborescence change). Dans ce cas, `GetParent` peut ne pas fonctionner (si le parent n'existe pas à cet endroit). Je présente dans ce livre plusieurs possibilités et ce sera à vous de choisir la meilleure option selon la situation.

Dans le cas de notre script, nous disposons déjà de l'instance de `Player` dans la variable `body` en paramètre de la méthode `_on_area_2d_body_entered(Node2D body)`. Du coup, il suffit de tester si l'instance `body` est de type `Player` et l'utiliser directement si c'est le cas. La méthode devient donc :

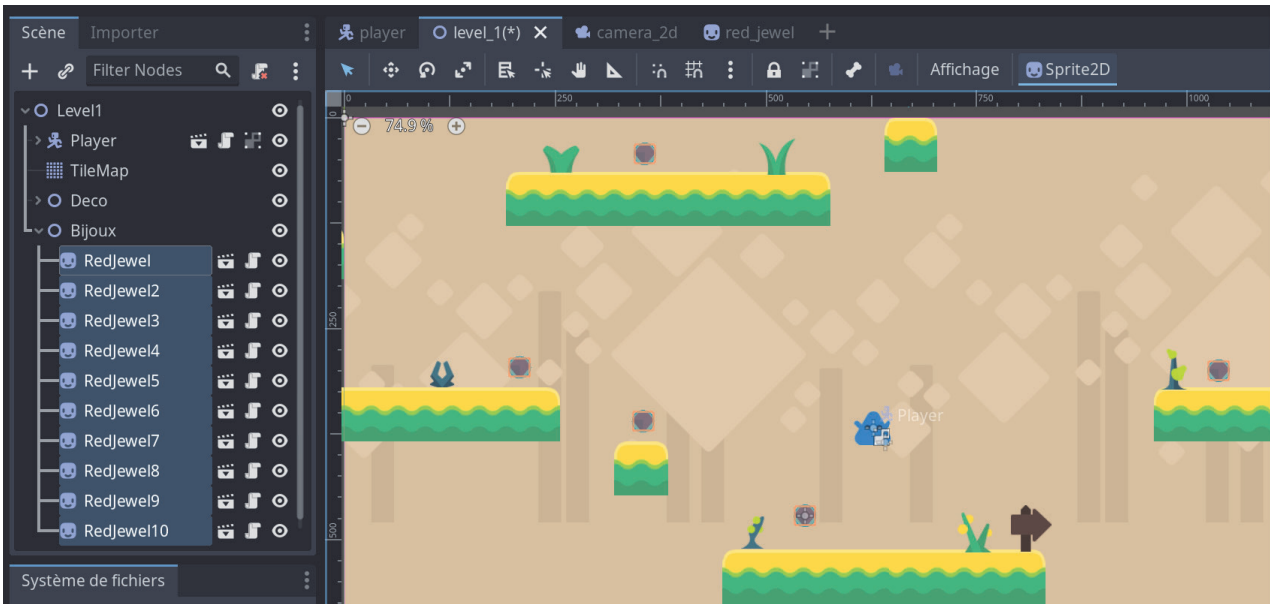
```
private void _on_area_2d_body_entered(Node2D body)
{
  if (body is Player)
  {
    (body as Player).coins ++;
    GD.Print("Nombre de pierres : " + (body as Player).coins);
    QueueFree();
  }
}
```

Et la référence `public player p;` n'est alors plus nécessaire. Lorsque vous maîtriserez tout le potentiel de Godot et de C#, vous pourrez utiliser la méthode avec laquelle vous êtes le plus à l'aise.

13.3. Test de notre script

Nous allons maintenant vérifier que notre système fonctionne correctement. Pour cela, sauvegardez votre travail, retournez dans la scène `level1` puis instanciez une pierre précieuse. Dupliquez-la une vingtaine de fois et disposez-les un peu partout dans votre niveau. Vous pouvez ranger ces pierres dans un Node2D afin d'organiser votre projet.

Figure 13.10 : Ajout des rubis à ramasser

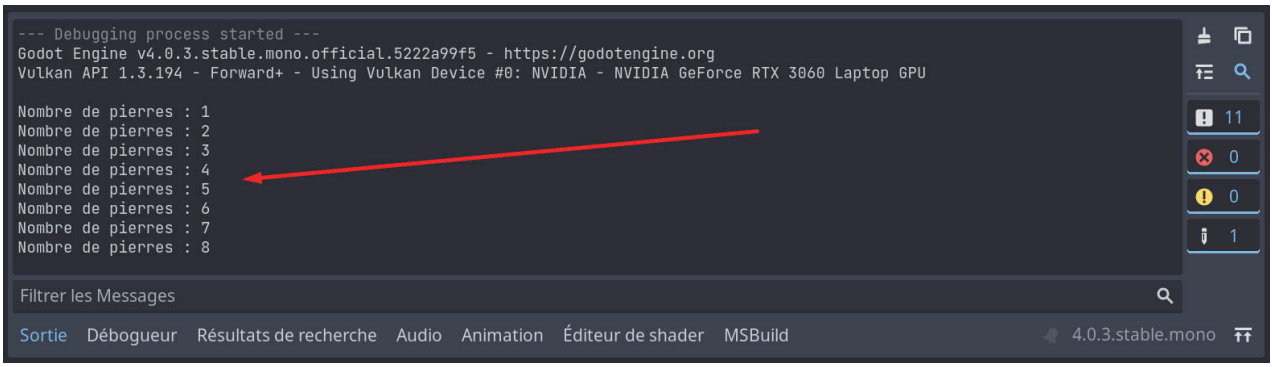


Attention › *Si vous réorganisez votre scène comme moi en positionnant les pierres dans un nœud, alors votre code ne fonctionnera plus. En effet, la fonction* `GetParent` *retourne le parent et si les pierres sont dans un nœud alors le parent sera ce nœud et pas le nœud principal. Pour remédier au problème, vous devez enchaîner deux* `GetParent` *pour récupérer le script du joueur :*

```
public override void _Ready()
  {
    p = GetParent().GetParent().GetNode<player>("Player");
  }
```

Une fois que tout est bien configuré, vous pouvez tester votre jeu. Déplacez-vous, ramassez les pierres et vérifiez qu'elles disparaissent comme prévu. Regardez également votre console pour vérifier que votre inventaire est bien mis à jour.

Figure 13.11 : *Le nombre de rubis dans l'inventaire*



Vous pouvez maintenant utiliser cette valeur pour déclencher des événements comme par exemple déclencher la fin du niveau si le nombre de pierres est supérieur à 10. Vous pouvez aussi utiliser ce système pour mettre en place une porte qui s'ouvre si le joueur possède une clé. Pour tester par le code si le joueur a une clé ou un nombre de rubis suffisant, vous pourrez utiliser une condition comme celle-ci :

```
if (p.coins >= 10)
{
  // Faire quelque chose
}
```

Pensez à supprimer la clé de l'inventaire une fois qu'elle a été utilisée. Idem, pour les pierres, vous pouvez les utiliser par exemple pour débloquer des choses ou pour gagner une vie tous les 10 rubis ramassés.

Dans le chapitre suivant, nous allons créer l'interface utilisateur pour afficher des informations à l'écran.

14

Création de l'interface utilisateur

Dans le chapitre précédent, nous avons ajouté la possibilité pour le joueur de ramasser des objets. Pour notre exemple, il s'agit de pierres précieuses. En tant que développeur, nous avons accès à la console de débogage, ce qui nous permet de tester notre système et de constater que celui-ci fonctionne puisque nous voyons le nombre de pierres ramassées par le personnage. Cependant, bien que notre système fonctionne, le joueur final lui n'a aucun moyen de savoir si tout a bien fonctionné ni même combien de pierres il possède. Nous allons donc mettre en place une GUI (Graphical User Interface/Interface graphique) qui affichera le nombre de rubis qu'a le joueur.

Nous mettrons en place une GUI classique comme l'on peut voir dans la plupart des jeux. C'est le cas par exemple dans Yo Frankie, un jeu libre et gratuit développé avec Blender. Sur la Figure 14.1, vous pouvez voir le nombre d'os du personnage car celui-ci peut s'en servir comme projectiles et les lancer.

Figure 14.1 : *Exemple de GUI du jeu Yo Frankie!*

14.1. Mise en place de l'interface

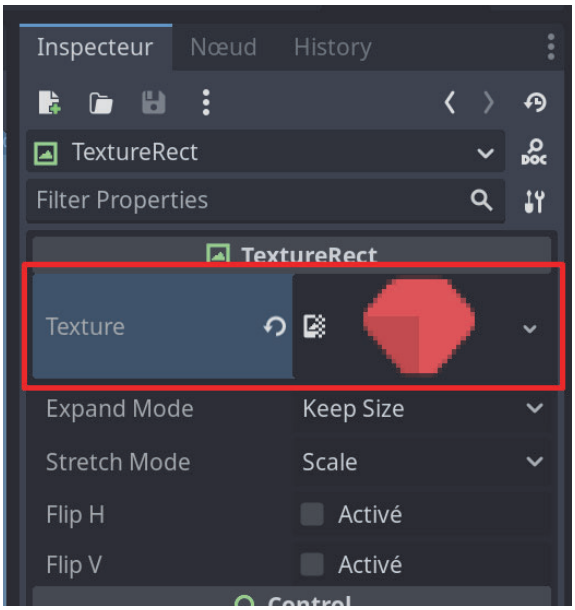
Nous allons construire notre interface utilisateur dans une nouvelle scène que je vous invite à créer.

Une fois la scène créée, ajoutez un nœud de type CanvasLayer. Il s'agit du nœud de base pour une interface graphique. Si vous connaissez Unity, vous ferez le rapprochement avec le composant Canvas. C'est dans ce nœud que nous allons construire l'interface utilisateur. Avant tout, renommez ce nœud en `GUI` et sauvegardez la scène dans le dossier des scènes.

À partir de là, nous allons pouvoir ajouter des éléments à notre interface. Nous commencerons par ajouter une image qui représentera les pierres précieuses. À côté de cette image, nous aurons le chiffre représentant le nombre de pierres récoltées. En voyant l'icône, le joueur doit comprendre directement de quoi il s'agit.

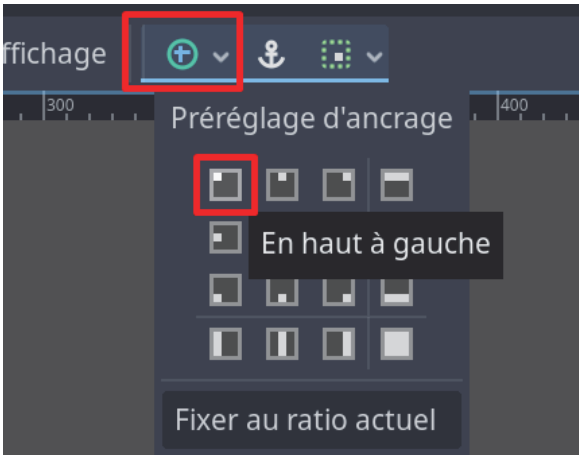
Ajoutez un nœud de type TextureRect. Ce composant permet, comme son nom l'indique, d'ajouter une texture à notre interface, dans notre cas ce sera la pierre rouge.

Figure 14.2 : *Ajout d'une texture*



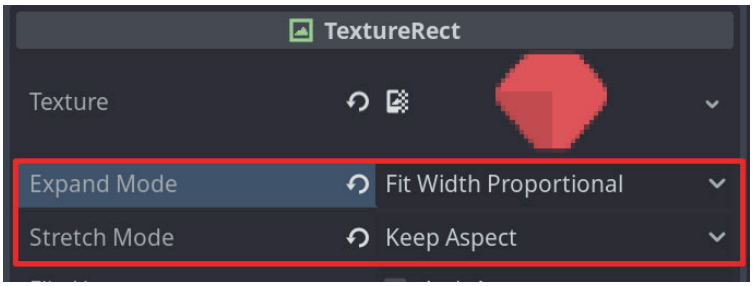
Nous allons ancrer cette texture en haut à gauche de notre écran. Pour cela, nous utilisons l'outil d'ancrage qui nous aidera à conserver une bonne position indépendamment de la taille de la fenêtre de jeu. Il est accessible dans le viewport depuis la barre d'outils supérieure.

Figure 14.3 : *Disposition à l'écran*



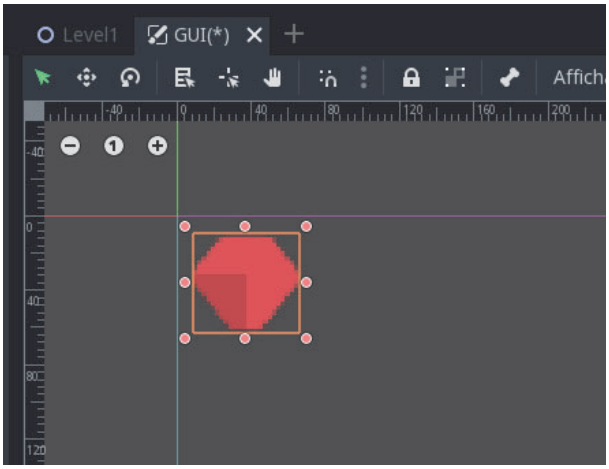
Au niveau de la disposition sur l'écran, choisissez la position EN HAUT À GAUCHE. Nous devons également modifier le EXPAND MODE ainsi que le STRETCH MODE via l'inspecteur afin de pouvoir agrandir la texture tout en conservant ses proportions. Choisissez respectivement FIT WIDTH PROPORTIONAL et KEEP ASPECT pour ces réglages :

Figure 14.4 : *Taille de l'image*



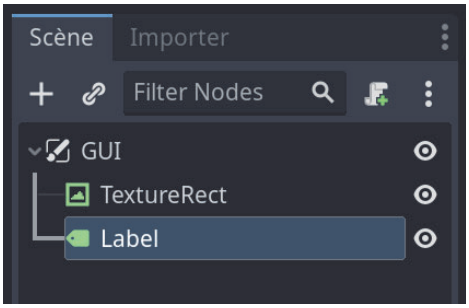
Placez ensuite précisément votre texture sur l'écran comme vous le souhaitez et choisissez une taille convenable.

Figure 14.5 : *Position de l'image*



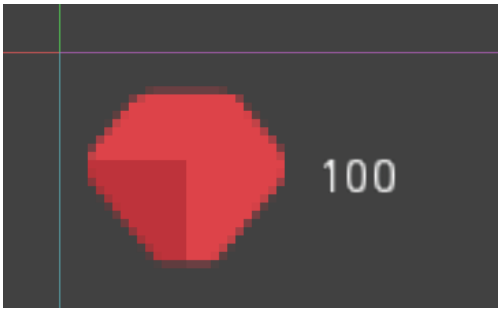
Nous allons maintenant prévoir une place pour un texte qui indiquera au joueur le nombre de pierres ramassées. Le texte sera un Label. Créez donc ce nœud Label.

Figure 14.6 : *Ajout du Label*



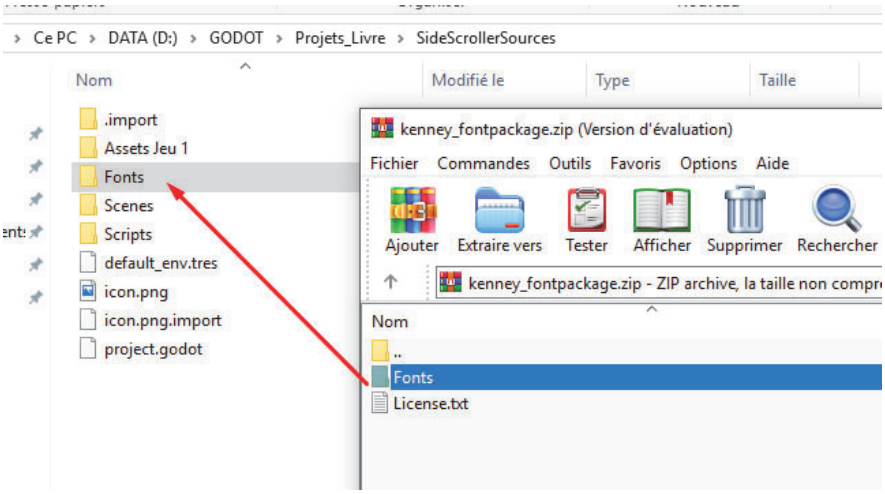
Positionnez le label en l'ancrant en haut à gauche. Placez le texte à droite de l'image et mettez un texte par défaut via l'inspecteur afin d'avoir une idée du rendu.

Figure 14.7 : *Placement du texte*



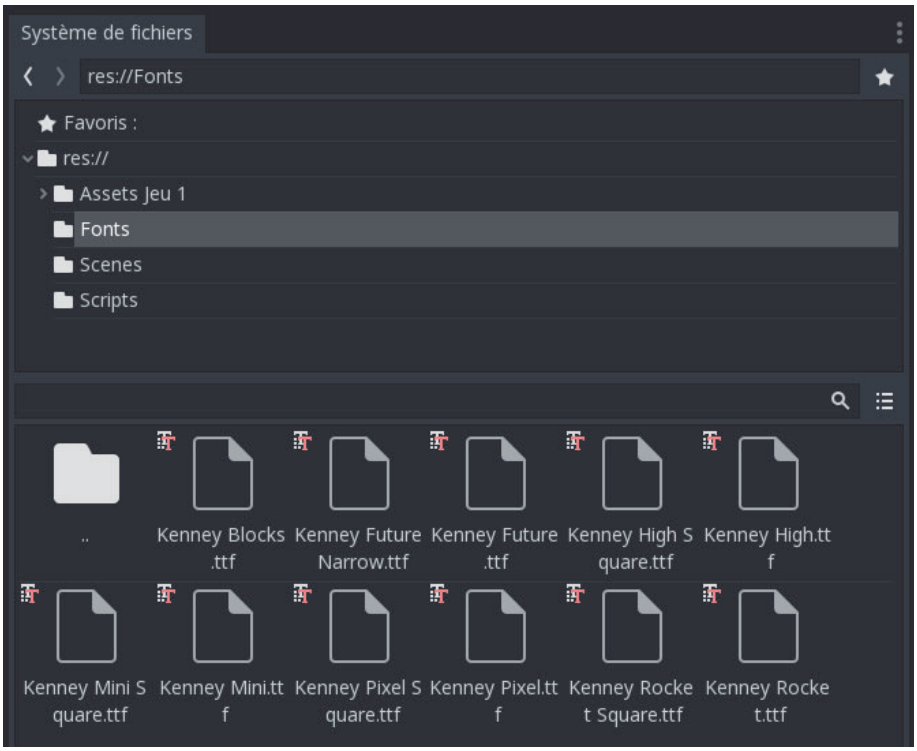
Notre interface commence à prendre forme ; cependant notre texte est beaucoup trop petit, nous devons le personnaliser. Si vous avez regardé l'inspecteur, vous avez vu que nous n'avons pas grand-chose sous la main pour personnaliser le texte. Nous allons donc télécharger une police d'écriture, par exemple une police gratuite via le site de Kenney. Le fichier ZIP téléchargé, une fois décompressé, vous donnera accès à plusieurs fichiers de police d'écriture. Importez le dossier `Fonts` dans le dossier des ressources du projet Godot. Pour cela, faites un clic droit sur le dossier `res` du système de fichiers puis cliquez sur MONTRER DANS LE GESTIONNAIRE DE FICHIERS. Copiez ensuite le dossier `Fonts`.

Figure 14.8 : *Import des polices*



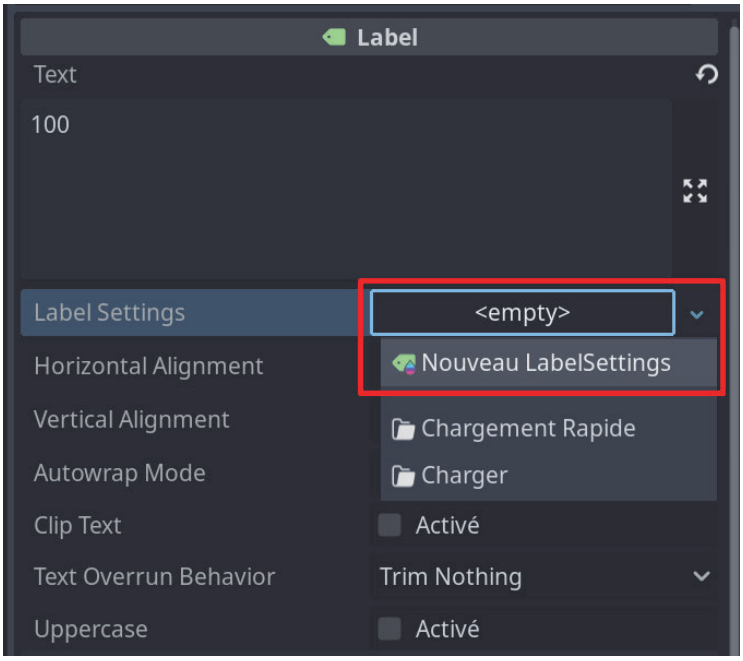
Retournez ensuite dans Godot. Si tout s'est bien passé, vous devriez avoir un nouveau dossier disponible :

Figure 14.9 : *Nos polices*



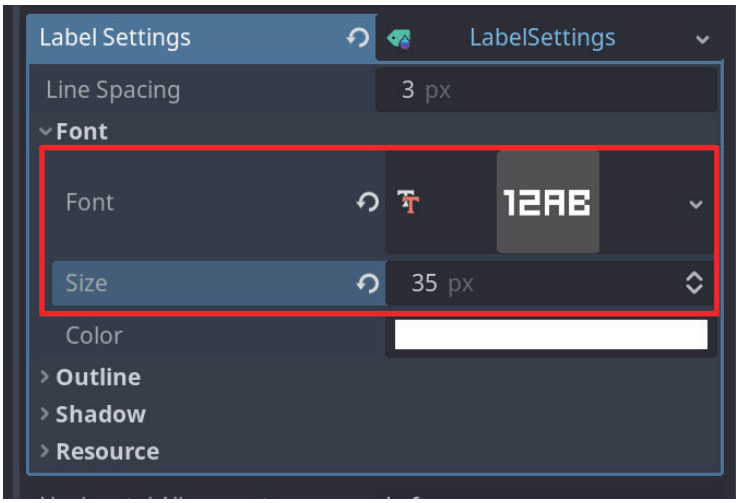
Maintenant que nous avons importé les polices d'écriture, nous allons pouvoir personnaliser le texte de notre interface utilisateur. Sélectionnez le label et consultez l'inspecteur. Ici nous créer un nouveau LabelSettings qui nous permettra de personnaliser la police d'écriture.

Figure 14.10 : *LabelSettings*



Dépliez la propriété FONT du LabelSettings. Dans le champ du paramètre FONT, vous avez la possibilité de glisser une police d'écriture. Pour ma part, j'ai choisi la police `Kenney Mini Square`. Vous pouvez aussi ajuster la taille des caractères (paramètre SIZE) à 35.

Figure 14.11 : *Personnalisation de la police d'écriture*



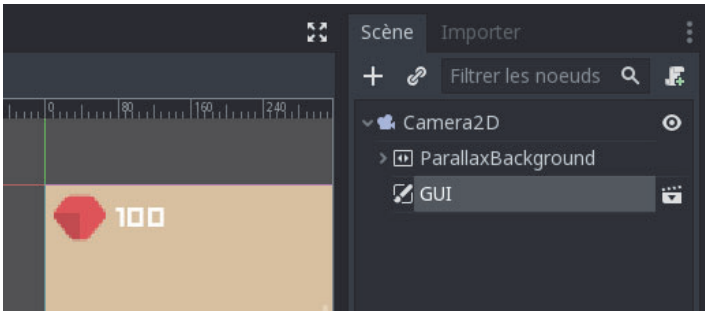
Ajustez les paramètres selon vos goûts. Le visuel en sera impacté.

Figure 14.12 : *GUI personnalisée*

Cela sera largement suffisant pour nos besoins. Nous allons maintenant voir comment implanter cette interface utilisateur dans notre jeu et comment la lier aux variables pour mettre à jour la valeur affichée.

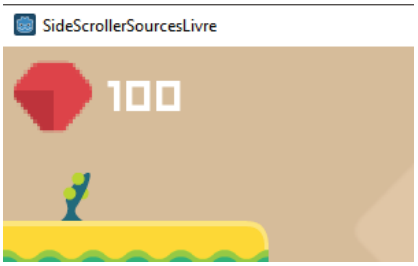
Sauvegardez la scène de votre GUI puis retournez sur la scène de la caméra. Dans cette scène, instanciez votre GUI en tant qu'enfant de votre caméra. Vous devriez avoir une scène caméra similaire à celle de la Figure 14.13.

Figure 14.13 : *Instanciation de la GUI*



Sauvegardez et retournez dans votre premier niveau, puis lancez le jeu pour vérifier que votre interface utilisateur est bien visible.

Figure 14.14 : *Notre GUI*



Nous avons réussi à mettre en place notre GUI. Nous allons maintenant programmer celle-ci.

14.2. Programmation de l'interface

À chaque fois que le joueur ramassera un objet, nous mettrons à jour notre interface afin d'afficher la bonne valeur (le bon nombre de pierres précieuses). Pour réaliser cela, nous allons créer un script sur notre GUI qui se chargera d'actualiser le texte puis nous appellerons cette fonction depuis le script qui gère les collisions. À chaque collision avec une pierre, la GUI se mettra à jour.

Retournez dans la scène `GUI` puis créez un nouveau script sur le nœud principal. Ce script sera donc placé sur le composant principal `GUI` (le CanvasLayer).

Nous allons commencer par mettre la valeur à zéro au lancement du jeu. Pour cela, accédez à la propriété texte du Label pour lui attribuer la valeur "0".

```
using Godot;
using System;

public partial class gui : CanvasLayer
{
  Label label;

  public override void _Ready()
  {
    label = GetNode("Label") as Label;
    label.Text = "0";
  }
}
```

Ce bout de code permet de récupérer le label, de le stocker dans une variable et de modifier la valeur de la propriété texte. Le tout se passe dans `ready` afin que le texte soit initialisé au lancement du jeu.

Nous créons maintenant la fonction qui nous permettra de mettre à jour l'interface utilisateur. Nous l'appellerons `ChangeVal`. Elle prend un paramètre `val` qui sera le nombre à afficher à l'écran.

```
public void ChangeVal(int val)
{
  label.Text = val.ToString();
}
```

Le paramètre `val` est donc la valeur passée à cette fonction et que nous voulons attribuer au texte. Nous utilisons la fonction `ToString` qui permet de convertir un nombre en texte car le Label attend un texte et pas une valeur numérique, nous devons donc convertir cette valeur de cette façon.

Retournez maintenant dans le script de collision de votre pierre précieuse. Ici, nous allons modifier la fonction de collision afin d'appeler la fonction `ChangeVal` pour mettre à jour le Label. Pour faire cela, commençons par créer une référence vers le script GUI :

```
public gui GUIScript;
```

Une fois cette variable créée, il faut lui attribuer une valeur. Cette variable doit contenir le script `gui.cs`. Pour récupérer ce script et le stocker dans la variable, procédez de cette façon :

```
public override void _Ready()
{
  p = GetParent().GetParent().GetNode<player>("Player");
  GUIScript = GetParent().GetParent().GetNode("Player/Camera2D/GUI") as gui;
}
```

Nous avons déjà vu comment récupérer le joueur. Pour récupérer le script `gui` c'est sensiblement la même chose à ceci près que nous devons spécifier tout le chemin pour atteindre le nœud souhaité. De plus, afin de vous montrer une autre façon de procéder, j'ai utilisé la notation `as gui` pour indiquer que je souhaite récupérer quelque chose de type GUI. vous pouvez désormais appeler la fonction permettant de mettre à jour votre GUI de cette façon :

```
GUIScript.ChangeVal(p.coins);
```

Cela nous donnera le script complet suivant :

```
using System;

public partial class RedJewel : Sprite2D
{
  public player p;
  public gui GUIScript;

  public override void _Ready()
  {
    p = GetParent().GetParent().GetNode<player>("Player");
    GUIScript = GetParent().GetParent().GetNode("Player/Camera2D/GUI") as
 gui;
  }

  private void _on_area_2d_body_entered(Node2D body)
  {
    p.coins ++;
    GD.Print("Nombre de pièces : " + p.coins);
    GUIScript.ChangeVal(p.coins);
```

```
            QueueFree();

            if (p.coins >= 10)
            {
              // Faire quelque chose
            }
        }
    }
```

Sauvegardez puis testez votre `Level1`. Ramassez quelques pierres, quatre par exemple et regardez le résultat :

Figure 14.15 : *Mise à jour de notre GUI*

Nous avons réussi à mettre en place notre GUI. Tout se met bien à jour comme attendu !

Pour vous entraîner et mettre en pratique vos connaissances, je vous invite à vous lancer dans un petit exercice. Développez un système de vies affiché à l'écran. Au bout de 100 rubis, le joueur gagne 1 vie. Vous pouvez également créer d'autres systèmes comme des clés à ramasser ou afficher le nom du niveau en cours, etc. Voici un exemple d'interface que vous êtes parfaitement en mesure de réaliser :

Figure 14.16 : *Exemple d'interface*

Afin que tout le monde travaille sur la même base, je ne garderai que l'interface affichant le nombre de pierres dans la suite de ce livre.

15

Ajout des ennemis

Dans ce chapitre, nous allons voir comment ajouter des pièges, c'est-à-dire principalement des ennemis qui apporteront de la difficulté et du challenge. Nous allons créer deux types d'ennemis qui seront suffisants pour la plupart de vos jeux 2D. Le premier ennemi sera basique, une simple boule de piques qui blessera le joueur si celui-ci le touche. Le second se déplacera de gauche à droite et le joueur perdra une vie s'il touche la tête de l'ennemi car celui-ci disposera d'un pique sur la tête. Nous aurons donc des ennemis statiques et des ennemis mouvants dans notre niveau. Nous mettrons aussi en place un élément qui servira à recharger le niveau si le joueur tombe dans le vide. Cet élément sera identique à l'ennemi fixe.

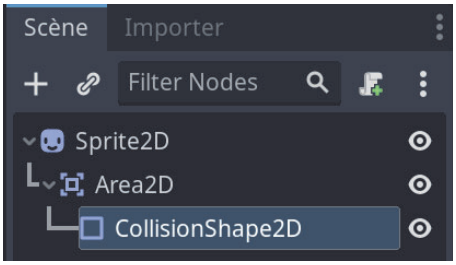
Par simplicité et pour ne se concentrer que sur l'essentiel, le système que nous allons mettre en place sera le suivant : lorsque le joueur touchera un ennemi ou un piège quel qu'il soit, le niveau se relancera depuis le début. Si vous le souhaitez, vous pourrez mettre en place un système de vies avec trois vies et un rechargement du niveau au bout du troisième contact. Vous avez toutes les clés pour développer un tel système car vous savez comment stocker une variable, mettre à jour un GUI et utiliser la condition `if`.

15.1. Création d'un ennemi statique

Nous allons commencer par créer l'ennemi statique car il est très simple à mettre en place. En effet, sa création est similaire à celle de la pierre précieuse. Créez une nouvelle scène sur laquelle nous allons pouvoir créer un ennemi.

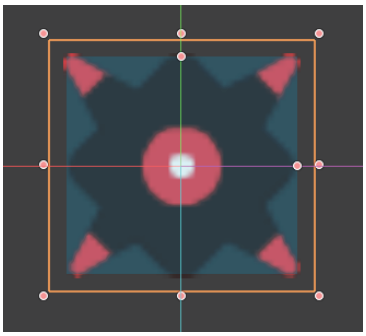
Créez un arbre similaire à la pierre, c'est-à-dire avec un Sprite2D puis un Area2D et enfin un CollisionShape2D qui aura un Shape de type rectangle. J'ai appelé l'ennemi `Spiky` (*pointu* ou *avec des pointes* en anglais) ce qui correspond à l'ennemi que nous sommes en train de créer.

Figure 15.1 : *Création de Spiky*



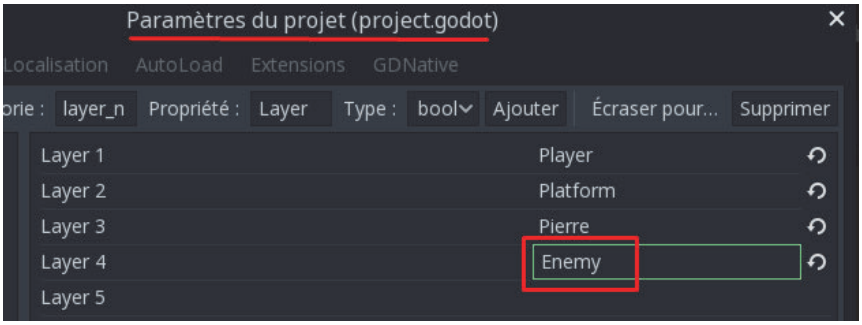
Une fois vos composants créés, configurez-les. Ajoutez une texture au Sprite afin de mettre en place le visuel et configurez le Shape afin de bien englober l'ennemi. Sur la Figure 15.2, vous pouvez voir le monstre ainsi que le CollisionShape que vous devez mettre en place. Je passe rapidement sur ces étapes qui sont exactement les mêmes que précédemment. Si vous avez des difficultés, relisez le chapitre Interaction avec les objets.

Figure 15.2 : *Spiky*



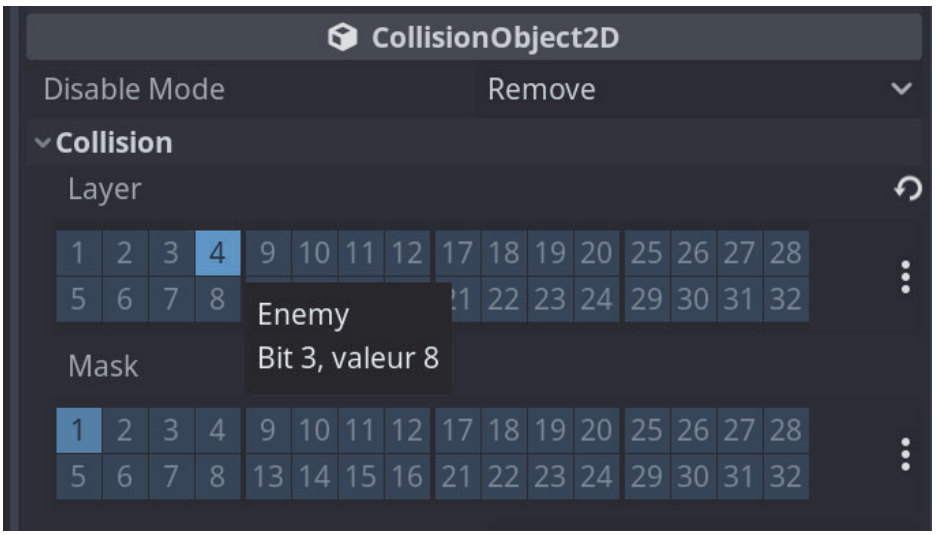
Nous allons maintenant créer un nouveau layer dans les options du projet où nous pourrons placer tous les ennemis que nous allons créer. Retournez donc dans PROJET/PARAMÈTRES DU PROJET puis dans le menu PHYSIQUE 2D ajoutez un nouveau calque [layer] que vous appellerez Enemy.

Figure 15.3 : *Création du layer Enemy*



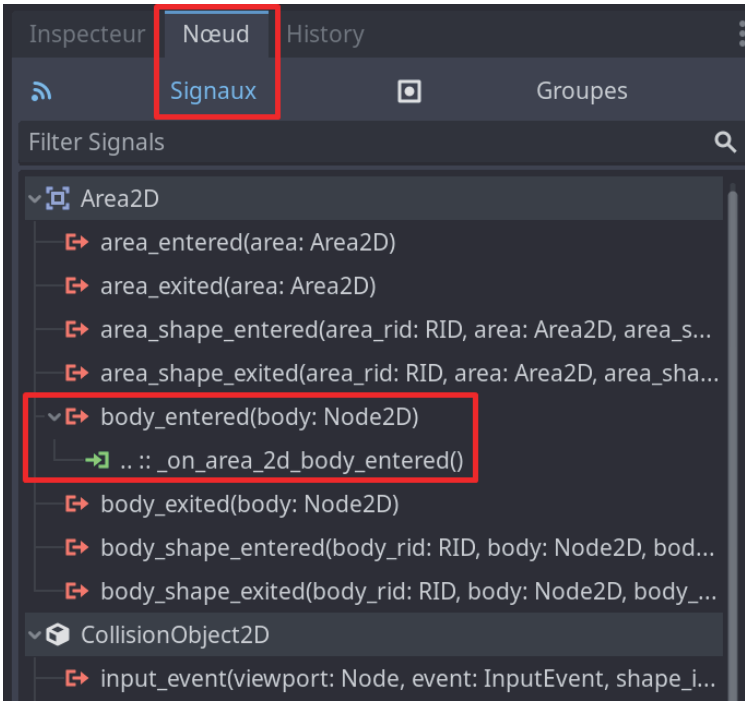
Maintenant que le layer est créé, placez-y votre ennemi en cliquant sur son Area2D puis en paramétrant correctement son LAYER sous la rubrique COLLISION. Cette manipulation nous permettra de gérer les collisions entre l'ennemi et le joueur.

Figure 15.4 : *On met l'ennemi sur son layer*



Une fois que tout est prêt, cliquez sur le Sprite afin de créer un nouveau script qui lui sera associé. Enregistrez-le dans le dossier des scripts et appelez-le `Spiky`. Maintenant que le script est créé, nous pouvons cliquer sur l'Area2D et connecter un signal au script du Sprite parent. Sélectionnez l'Area2D, parcourez la liste des nœuds et connectez le signal `body_entered`.

Figure 15.5 : *Création du signal*



Une fois le signal connecté, retournez dans votre script `Spiky`. Comme indiqué dans l'introduction de ce chapitre, au premier contact entre le joueur et l'ennemi, nous relancerons la scène. Pour recharger la scène courante, nous utilisons la fonction `GetTree().ReloadCurrentScene()`. Le code complet de votre ennemi sera le suivant :

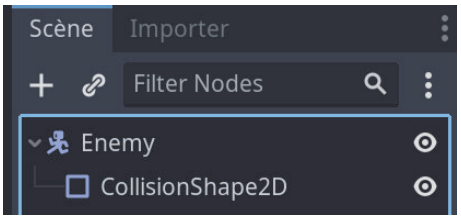
```
using Godot;
using System;

public partial class spiky : Sprite2D
{
  private void _on_area_2d_body_entered(Node2D body)
  {
    GetTree().ReloadCurrentScene();
  }
}
```

Votre premier ennemi est créé ! Sauvegardez cette scène afin de la garder au chaud. Nous allons directement passer à la création du second ennemi qui se déplacera de gauche à droite de façon automatique.

15.2. Création d'un ennemi mouvant

Créez une nouvelle scène pour cet ennemi. Le nœud principal sera un CharacterBody2D que vous renommerez en `Enemy`. Pour fonctionner, ce CharacterBody2D aura besoin d'un CollisionShape2D qui lui-même aura besoin d'un Shape de type rectangle. Créez donc ces éléments.

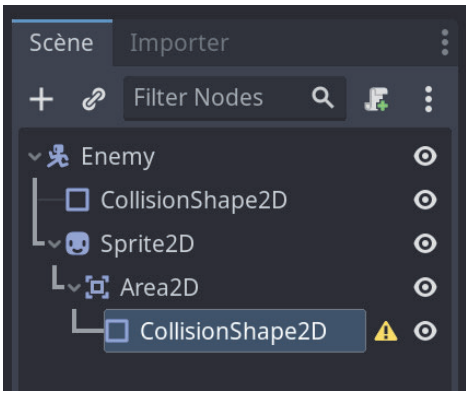
Figure 15.6 : *Création du second ennemi*



Pour cet ennemi, nous utiliserons deux CollisionShape2D. L'un servira à gérer les collisions du monstre avec le décor ; le second, placé sur sa tête (le pique), servira à blesser le joueur.

Nous allons maintenant créer un nouveau nœud qui sera un Sprite2D. Ce Sprite aura besoin d'un Area2D en enfant et d'un CollisionShape2D en sous-enfant ce qui nous donne la structure complète exposée à la Figure 15.7.

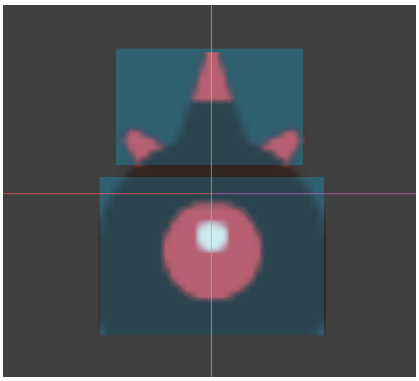
Figure 15.7 : *Structure de l'ennemi*



Nous avons tout ce qu'il nous faut. Ajoutez une texture, choisissez-la dans le dossier `Enemies` des assets.

Configurez ensuite les CollisionShape2D. Pour le premier, créez un rectangle de la largeur de l'ennemi sans couvrir la tête. Le second Shape sera aussi un rectangle mais uniquement placé sur la tête. La Figure 15.8 vous aidera à comprendre ce que nous devons faire. La collision au niveau de la tête (des piques) sera pour attaquer le joueur, la collision au niveau du corps (en dessous) sera utilisée pour détecter la collision avec les murs.

Figure 15.8 : *Mise en place des Shape*



La chose importante ici est de veiller à ce que les deux colliders du monstre ne se touchent pas car dans le cas contraire cela pourrait provoquer un comportement non désiré. Pensez également à mettre votre ennemi sur le calque des ennemis comme pour `Spiky`.

Créez un script sur votre ennemi au niveau de son CharacterBody2D. Une fois créé, enregistrez-le puis allez sur l'Area2D afin de connecter le signal `body_entered`. Écrivez ensuite le code permettant de relancer le niveau comme nous l'avons fait pour `Spiky`.

```
using Godot;
using System;

public partial class enemy : CharacterBody2D
{
  private void _on_area_2d_body_entered(Node2D body)
  {
    GetTree().ReloadCurrentScene();
  }
}
```

Notre ennemi est maintenant identique au premier. Nous allons lui ajouter la fonctionnalité de mouvement allant de gauche à droite. Pour créer cette animation, nous allons écrire un bout de code dans la fonction `_PhysicsProcess` qui tourne en boucle (s'exécute 60 fois par seconde si le jeu tourne en 60 fps).

Nous utiliserons la fonction `MoveAndCollide` pour déplacer l'ennemi et gérer les collisions avec les murs. Nous passerons en paramètre une vitesse de déplacement et, s'il y a collision, nous ferons rebondir l'ennemi (en lui faisant prendre une direction opposée) avec la fonction `bounce`. Pour récupérer la direction opposée, nous avons à disposition la propriété `normal` de la collision détectée.

Commencez par créer une variable de type `Vector2` qui stockera la vitesse de déplacement :

```
Vector2 velocity = new Vector2(150,0);
```

Appliquez ensuite le mouvement tout en récupérant les informations de collision. Comme le paramètre `delta` de la fonction `_PhysicsProcess` est de type `double`, nous devons le convertir en `float`.

```
var collision = MoveAndCollide(velocity * (float)delta);
```

Détectez s'il y a une collision à l'aide d'une condition :

```
if (collision != null)
{

}
```

Enfin, appliquez le rebond :

```
velocity = velocity.Bounce(collision.GetNormal());
```

Le code complet de notre ennemi est donc le suivant :

```
using Godot;
using System;

public partial class enemy : CharacterBody2D
{
  Vector2 velocity = new Vector2(150,0);

  public override void _PhysicsProcess(double delta)
  {
    var collision = MoveAndCollide(velocity * (float)delta);

    if (collision != null)
    {
      velocity = velocity.Bounce(collision.GetNormal());
    }
  }

  private void _on_area_2d_body_entered(Node2D body)
  {
    GetTree().ReloadCurrentScene();
  }
}
```

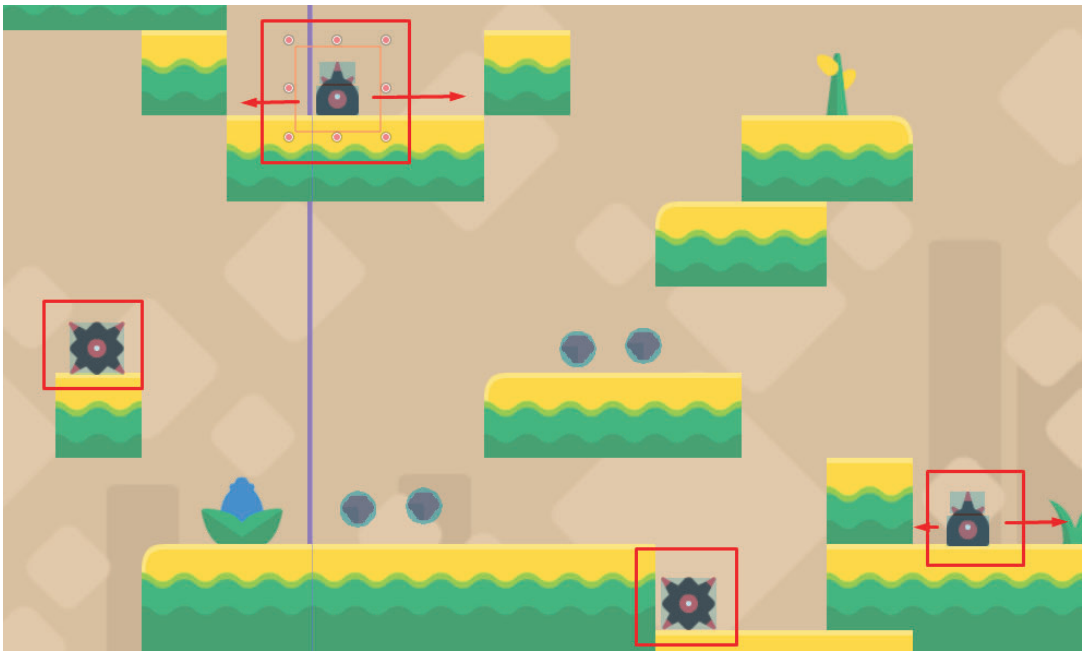
Attention › *Dans mon exemple, le collider sur la tête est plus petit que le collider du corps. Cela signifie que lorsque le monstre entrera en collision avec un mur, c'est son corps qui touchera le mur pas sa tête. Si la tête entre en collision avec un objet, alors le niveau se relance. Donc si le collider de la tête entre en collision avec un mur, le jeu se relancera. Normalement seul le joueur peut toucher la tête du monstre mais si vous voulez plus de sécurité alors vous pouvez ajouter le test suivant afin d'être certain que seul le joueur sera pris en compte lors d'une collision avec la tête :*

```
private void _on_area_2d_body_entered(Node2D body)
{
  if (body.Name == "Player")
  {
    GetTree().ReloadCurrentScene();
  }
}
```

Avec ce test, vous ne risquez pas d'avoir un comportement inatendu car avant de relancer le jeu, on vérifie si c'est bien le joueur qui a touché le monstre.

Enregistrez le script ainsi que la scène. Votre second ennemi est prêt ! Son comportement sera très simple : il ira de gauche à droite et fera demi-tour dès qu'il touchera un mur invisible.

Figure 15.9 : Ajout des ennemis à la scène



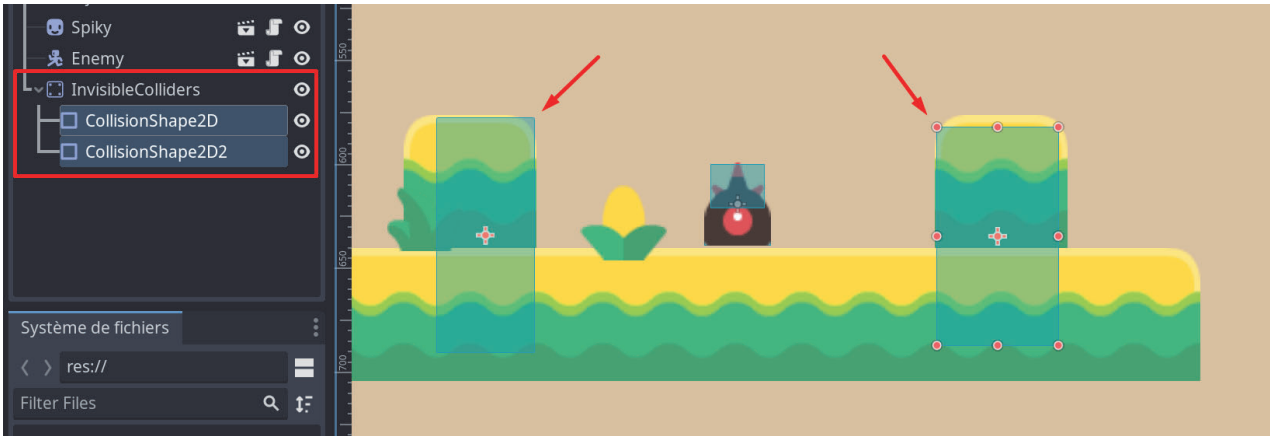
Nos ennemis sont ajoutés à la scène. Vous pouvez lancer le jeu afin de vérifier que les ennemis statiques restent en place, que les ennemis mouvants se déplacent bien horizontalement et que vous perdez bien si vous touchez Spiky ou le haut de la tête de l'ennemi mouvant. Si vos ennemis mouvants traversent les murs c'est normal, nous allons corriger cela dans un instant.

Figure 15.10 : Les ennemis en action

Attention *› Lorsque vous placez les ennemis sur le niveau, vérifiez que leurs CollisionShape2D ne soient pas dans un mur, mettez vos ennemis légèrement au-dessus du sol (1 ou 2 pixels en lévitation). Cela évitera d'éventuels comportements indésirables.*

Pour éviter que les ennemis ne traversent les tuiles, nous devons créer des murs invisibles. Nous créons un nœud de type StaticBody2D afin de mettre en place la collision entre le monstre et le mur. Ajoutez ensuite autant de CollisionShape2D que de murs invisibles nécessaires.

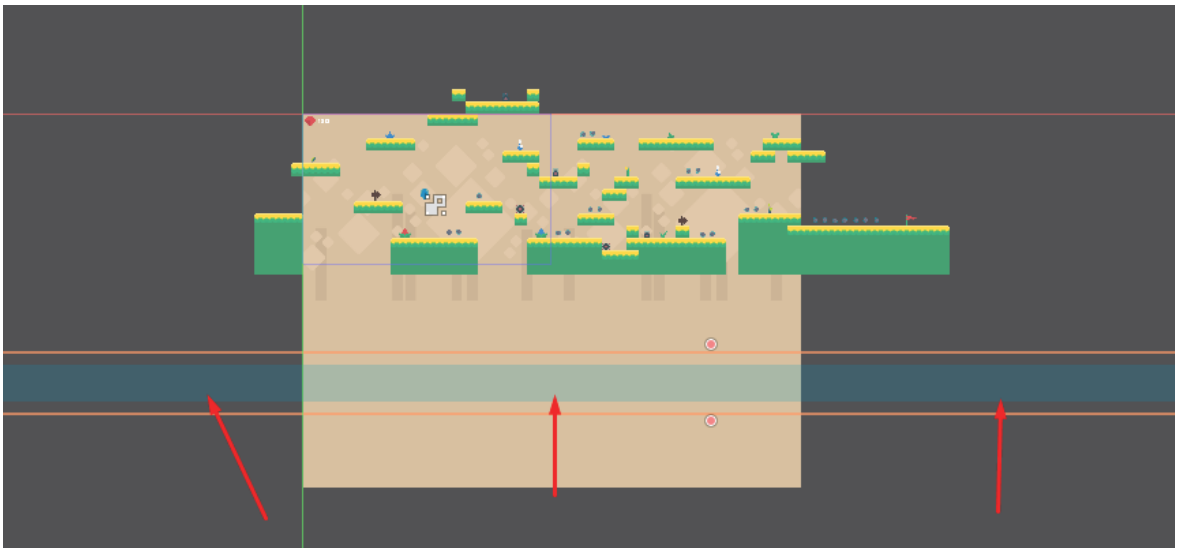
Figure 15.11 : *Les murs invisibles*



Vous pouvez de nouveau tester votre niveau. Si tout fonctionne bien, le niveau se relancera à chaque contact avec un ennemi. En revanche, le joueur peut tomber dans le vide. S'il tombe d'une plateforme, il chutera à l'infini. Ce qui peut poser problème. Nous allons remédier à cela.

Dupliquez un ennemi `Spiky` et placez-le en dessous du niveau. Supprimez sa texture via l'inspecteur et agrandissez l'objet (Propriété SCALE du TRANSFORM) pour couvrir toute la surface en dessous du niveau et être sûr que si le joueur tombe, il touchera forcément cet objet.

Figure 15.12 : *Collision avec le vide sous le niveau*



Tous les jeux utilisent ce système d'objet de collision immense au cas où le joueur tomberait sous le décor. Encore plus dans notre cas et dans le cas des jeux de plateformes, il arrivera forcément un moment ou le joueur tombera dans le vide. Désormais, si le joueur tombe, il touchera cet objet de collision et le niveau se relancera.

Vous pouvez mettre en pratique les connaissances acquises au cours de ce chapitre par exemple en créant un nouveau type d'ennemi qui sera le boss final du niveau avec un comportement un peu plus complexe comme par exemple une vitesse aléatoire. N'hésitez pas à vous amuser et à imaginer des éléments alternatifs à ce que je vous présente dans ce livre.

16

Musique et effets sonores

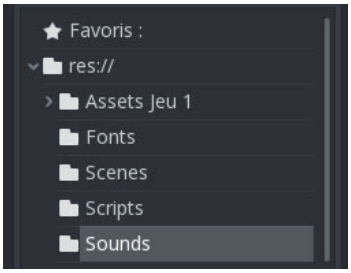
Dans le monde du jeu vidéo, le son est un élément très important. La musique et les effets sonores créent une ambiance dans votre jeu et renforce l'immersion. Cela permet au joueur de vivre une meilleure expérience ainsi que d'avoir des retours autres que visuels. Par exemple, lorsqu'un ennemi touche le personnage, vous pouvez le faire clignoter afin de montrer qu'il a été touché. Dans un combat avec un boss, le joueur doit regarder partout pour ne rien rater et le son peut être un très bon complément au visuel. Vous pouvez déclencher un effet sonore lorsque le joueur est touché ; ainsi, si le joueur ne voit pas l'effet visuel, il entendra l'effet sonore complémentaire.

Dans un jeu vidéo, toutes les actions sont accompagnées de sons : les bruits de pas, les bruits de saut ou de combat, la musique de fond, les clics sur les menus, etc. Nous allons découvrir comment mettre en place du son dans notre jeu avec Godot.

16.1. Musique d'ambiance

Nous allons commencer par ce qu'il y a de plus simple : la musique d'ambiance. Sélectionnez une musique qui vous plaît. Vous pouvez vous rendre sur opengameart.org pour trouver une musique de fond pour votre jeu. Nous en avons trouvé une qui a l'air de correspondre à notre jeu : `Town Theme` (de Ted Kerr). Téléchargez le fichier et importez-le dans Godot dans un dossier `Sounds` que vous aurez créé au préalable.

Figure 16.1 : Import de la musique

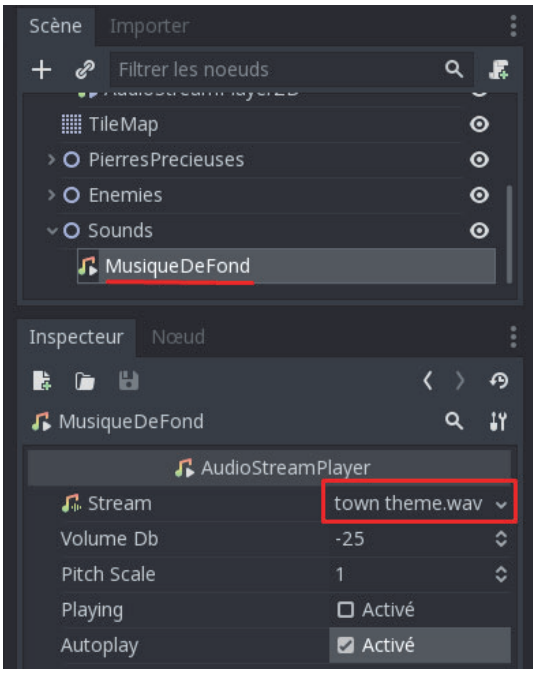


Attention *› Godot ne supporte pas tous les formats audio. Vous devez utiliser des fichiers au format WAV ou OGG. Si vous téléchargez une musique en MP3, vous aurez besoin de convertir cette musique en WAV pour qu'elle soit compatible avec Godot.*

Nous allons maintenant ajouter cette musique à notre premier niveau. Dans votre scène `Level1`, ajoutez un nœud vide que vous appellerez `Sounds` puis un nœud de type AudioStreamPlayer. Renommez-le à votre guise.

Une fois que vous avez mis en place l'AudioStreamPlayer, glissez/déposez votre musique dans l'emplacement STREAM prévu pour l'accueillir.

Figure 16.2 : *Ajout de la musique*



AUDIOSTREAMPLAYER

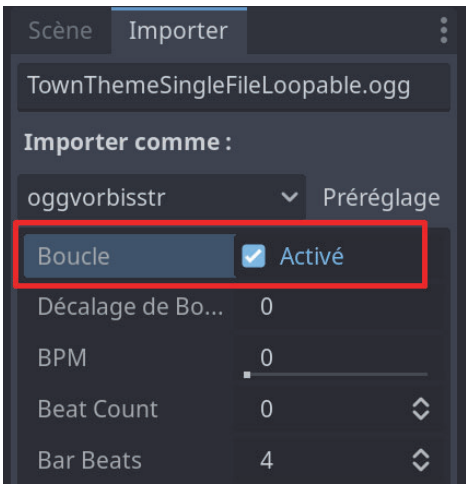
Le composant AudioStreamPlayer permet de jouer un son. Vous aurez accès à certaines options comme le volume du son, le fait qu'il doit être joué en boucle ou pas ou encore au fichier audio qui doit être joué par ce composant. Ce composant ne prend pas en compte l'espace. Cela signifie que le son sera le même partout dans la scène. Il n'y a pas de notion de distance par rapport à la source du son, c'est donc ce composant qu'il nous faut pour une musique d'ambiance.

Nous voulons créer une musique d'ambiance. Celle-ci doit donc se déclencher au lancement du jeu et tourner en boucle.

Pour faire cela, nous devons d'une part activer l'option AUTOPLAY de l'AudioStreamPlayer correspondant à notre musique de fond (voir Figure 16.2). C'est cette option qui déclenche la musique automatiquement. D'autre part, nous devons configurer notre fichier audio en mode Boucle. Dans le système de fichiers, cliquez sur votre fichier audio puis

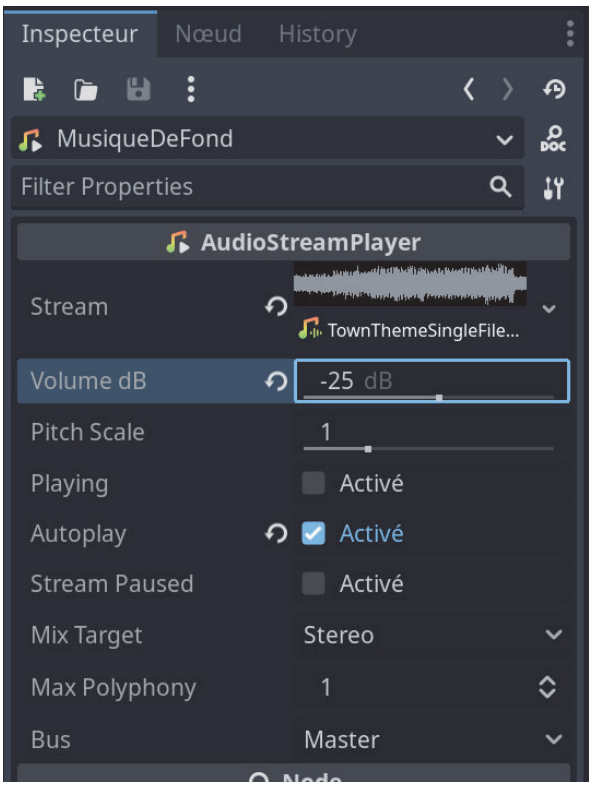
rendez-vous sous l'onglet IMPORTER. C'est là que vous trouverez l'option BOUCLE [Loop] qu'il vous faudra cocher.

Figure 16.3 : *Mode Boucle*



Cliquez sur RÉ-IMPORTER pour appliquer vos changements puis retournez sur votre AudioStreamPlayer afin de poursuivre sa configuration. Actuellement, le son est un peu trop fort pour une musique de fond. Vous pouvez diminuer le volume en mettant une valeur négative à l'option VOLUME DB.

Figure 16.4 : *Volume de la musique*



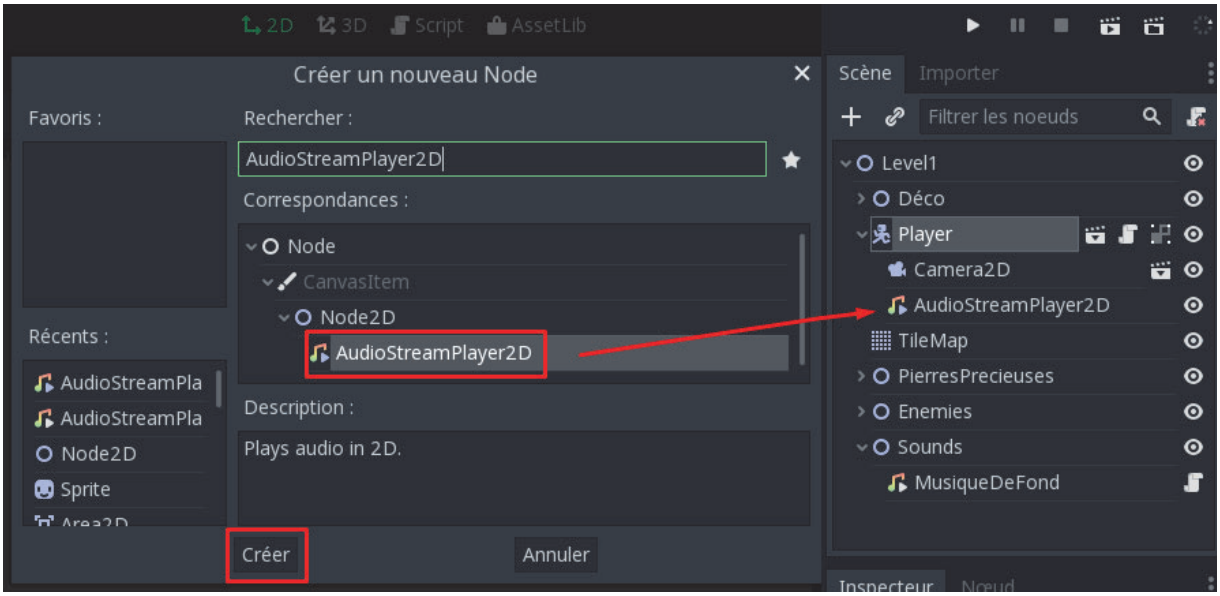
Lancez votre jeu pour vérifier que la musique se joue comme attendu. Comme vous pouvez le voir, la manipulation est très simple et ne requiert pas de programmation.

16.2. Effets sonores

Nous allons maintenant voir comment lancer des effets sonores par script. Pour notre exemple, nous émettrons un bruit de saut chaque fois que le personnage sautera et un bruit de pièce quand le personnage ramassera des rubis. Le son ne tournera donc pas en boucle, il sera déclenché par script lors d'un événement bien précis.

Pour commencer, nous allons avoir besoin d'ajouter un nouveau composant à notre scène. Ce composant sera un AudioStreamPlayer2D. Ajoutez-le en tant qu'enfant du Player dans votre niveau.

Figure 16.5 : *AudioStreamPlayer2D*



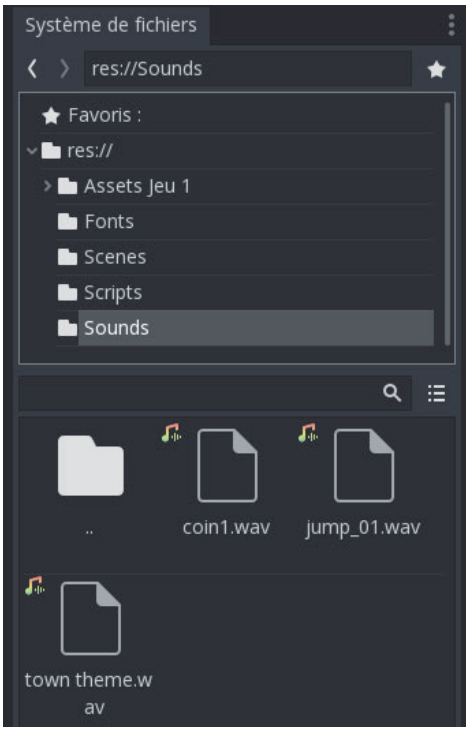
Un peu comme l'AudioStreamPlayer classique, ce composant va vous permettre d'émettre un son. Cependant, cette fois-ci, le son sera spatialisé en 2D. Dans notre cas, il n'y aura pas vraiment de différence mais sachez que le son sera émis à partir d'une source et qu'il sera légèrement différent selon l'endroit où se trouve le personnage et la direction vers laquelle il se déplace.

***Note** › Il existe une variante 3D du AudioStreamPlayer qui spatialise le son en 3D. Un personnage se trouvant loin de la source sonore entendra plus faiblement le son que s'il était juste à côté.*

Vous allez avoir besoin d'importer des sons dans votre projet. Je vous propose de récupérer un bruit de pièce et un bruit de saut. Si vous avez du mal à trouver des effets

sonores, voici ceux que j'ai utilisés : `Platformer Jumping Sounds` (de Devin Watson) et `10 8bit coin sounds` (de Luke RUST LTD). Importez-les dans votre projet.

Figure 16.6 : *Nos sons*



Retournez ensuite dans le script de votre personnage. Nous allons créer deux nouvelles variables qui vont nous servir à stocker les deux sons que nous utiliserons. En plus des sons, il nous faudra une variable pour stocker le lecteur audio. Créez donc une variable pour le son de la pièce, une variable pour le son de saut et une variable pour le lecteur audio :

```
AudioStream coinSound;
AudioStream jumpSound;
AudioStreamPlayer2D audioPlayer;
```

Dans la fonction `_Ready`, récupérez le lecteur audio avec la fonction `GetNode`. Renseignez également les variables des sons en chargeant le fichier audio avec la fonction `Load`. Cette fonction permet de charger une ressource à un chemin donné. Voici comment l'utiliser :

```
public override void _Ready()
  {
    audioPlayer = GetNode<AudioStreamPlayer2D>("AudioStreamPlayer2D");
```

```
        coinSound = GD.Load("res://Sounds/coin1.wav") as AudioStream;
        jumpSound = GD.Load("res://Sounds/jump1.wav") as AudioStream;

        // ... suite de la fonction
    }
```

Cela nous permet de charger les fichiers audio et de les associer à nos deux variables. Vous remarquerez que j'utilise `as AudioStream` afin de bien indiquer le type de la variable.

Attention *› Adaptez le chemin à votre cas. Vérifiez le nom du dossier ainsi que le nom du fichier audio que vous voulez charger. Si votre son de saut ne s'appelle pas `jump1.wav` vous devrez remplacer le chemin pour qu'il corresponde à votre projet. Un mauvais chemin empêchera le chargement de la ressource.*

La dernière chose à faire est de déclencher le son lorsque nous en avons besoin. Nous allons commencer par le saut. Lorsque le joueur appuiera sur la touche Espace, le personnage sautera et le son de saut se déclenchera. Pour accéder à l'AudioStreamPlayer2D par script, vous écrirez `audioPlayer` (car c'est comme cela que nous avons nommé notre variable). La fonction `Play()` permet de jouer le son. Pour le moment, le son n'a pas été associé au Player ; vous devez indiquer quel son doit être joué en modifiant sa propriété STREAM.

```
audioPlayer.Stream = jumpSound;
```

Il ne reste plus qu'à appeler la fonction `Play` :

```
audioPlayer.Play();
```

Tout ceci doit être écrit dans la condition qui gère le saut. Voici où vous devez placer ce code :

```
// Gestion du saut si on appuie sur espace
if (Input.IsActionJustPressed("ui_accept") && IsOnFloor())
  {
    velocity.Y = JumpVelocity;
    audioPlayer.Stream = jumpSound;
    audioPlayer.Play();
  }
```

Testez votre code en lançant votre jeu et en appuyant sur la touche Espace. Si tout va bien, le son devrait se déclencher.

Passons maintenant au son produit lorsque le personnage ramasse un rubis. La procédure est la même. Nous allons créer une fonction `playCoinSound` dans le script `Player` qui servira à déclencher le bruit de pièce :

```
public void playCoinSound()
  {
    audioPlayer.Stream = coinSound;
    audioPlayer.Play();
  }
```

La collision entre le personnage et la pierre est gérée dans le script `redJewel` si vous avez utilisé le nom par défaut. Allez donc dans le script de votre pierre et appelez la fonction `playCoinSound` fraîchement créée.

```
p.playCoinSound();
```

Ce qui nous donne la fonction complète suivante :

```
private void _on_area_2d_body_entered(Node2D body)
  {
    p.playCoinSound(); // Jouer le son
    p.coins ++;
    GD.Print("Nombre de pièces : " + p.coins);
    GUIScript.ChangeVal(p.coins);
    QueueFree();
  }
```

Lorsque le joueur ramasse une pierre, le programme exécute cette fonction, qui va elle-même appeler la fonction que nous venons de créer dans le script du personnage.

Testez en enregistrant et en lançant votre jeu. Ramassez une pierre précieuse et vérifiez que le son se déclenche comme convenu.

Vous avez donc vu comment faire tourner une musique d'arrière-plan en boucle et comment déclencher des effets sonores ponctuellement par script. Grâce à cela, vous êtes maintenant en mesure de mettre en place une ambiance sonore dans vos jeux.

17

Finalisation de notre jeu

Nous avons vu comment créer un jeu 2D à l'aide des différents outils proposés par Godot. Pour le rendre pleinement fonctionnel, il nous reste encore à lui ajouter un menu principal pour que le joueur puisse lancer une nouvelle partie. Puis nous devons compiler le jeu pour créer un exécutable que vous pourrez partager.

17.1. Création du menu principal

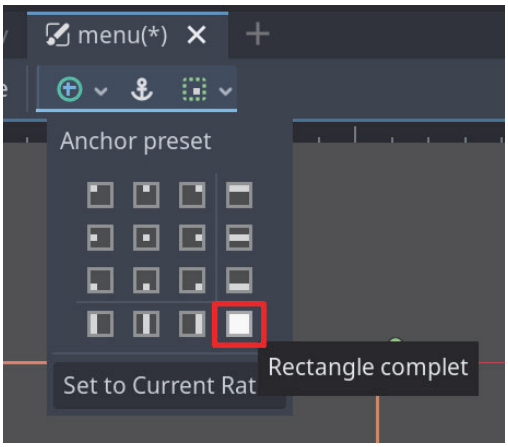
Nous allons commencer par concevoir le menu principal de notre jeu. Cette page d'accueil permettra au joueur de lancer le jeu ou de revenir sur son bureau. Notre interface sera très simple. Nous y mettrons : une image de fond, un texte avec le titre du jeu, un bouton JOUER et un bouton QUITTER.

Pour mettre en place ce menu, créez une nouvelle scène que vous appellerez `Menu`. Ajoutez-lui un CanvasLayer comme premier nœud. Le composant CanvasLayer dispose de la propriété LAYER [calque] qui a pour valeur un nombre entier. Celui-ci indique sur quel plan seront positionnés les nœuds enfants du CanvasLayer. Par défaut, pour un jeu 2D, les éléments sont affichés sur le calque 0. Un CanvasLayer est par défaut sur le calque 1. Cela signifie que le contenu sera affiché au-dessus du contenu du jeu. Cela est très pratique surtout pour les éléments comme la barre de vie ou le score. En effet, ces éléments doivent toujours être affichés à l'écran, il faut donc les mettre sur le calque 1 pour qu'ils soient au-dessus du jeu et donc visibles.

Dans ce CanvasLayer, vous pouvez, si cela vous est nécessaire créer des conteneurs. Les *conteneurs* ont pour fonction de *contenir* des éléments et éventuellement de les organiser. Si votre interface doit être composée de plusieurs blocs, vous utiliserez des conteneurs. Si vous avez besoin d'aligner des éléments, vous pourrez utiliser des conteneurs verticaux (VBoxContainer) ou horizontaux (HBoxContainer). Dans notre cas, nous allons rester très simple et mettre directement nos éléments d'interface.

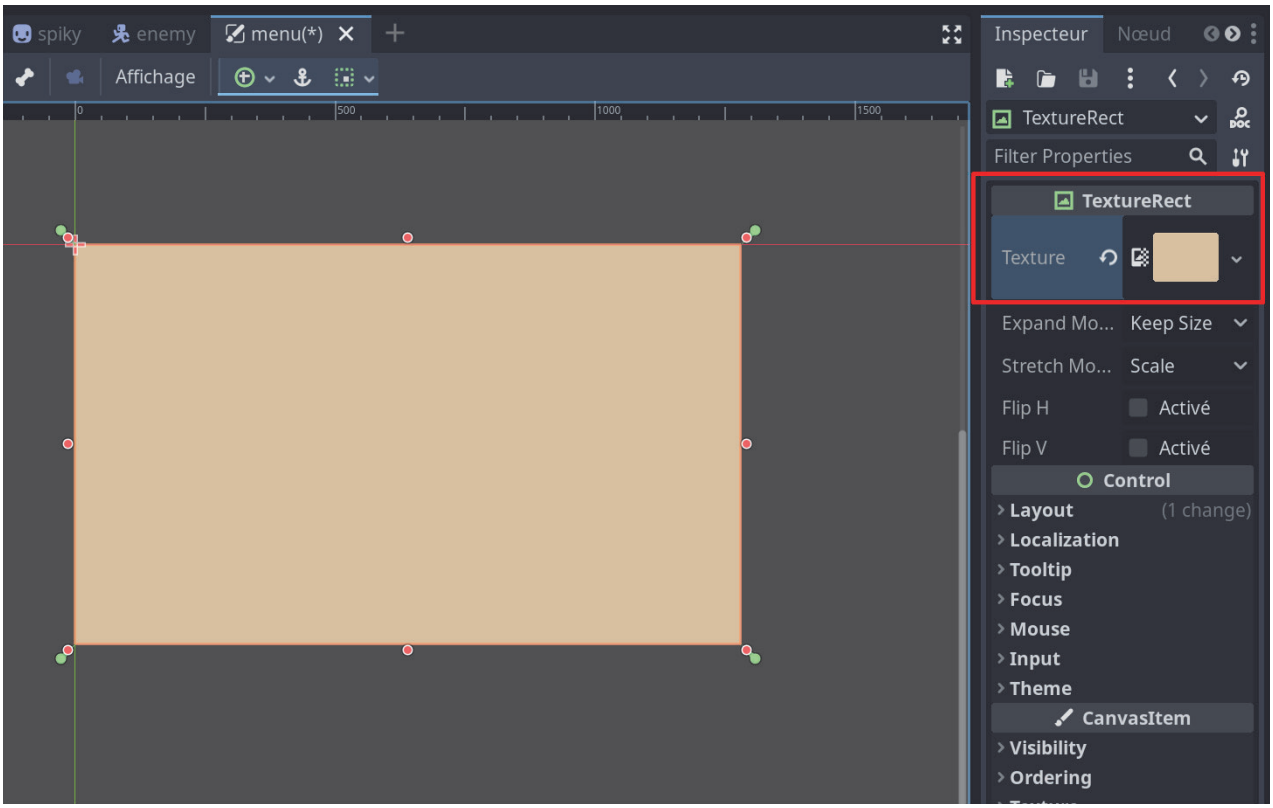
Commencez par ajouter un TextureRect. Cet élément permet d'appliquer une image de fond à notre menu. Nous débutons par le fond car l'ordre est important. Les premiers éléments sont dessinés en premier. Si vous placez d'abord vos boutons puis l'image de fond, celle-ci s'affichera au-dessus et donc l'interface disparaîtra. Ajustez la disposition sur l'écran de votre TextureRect sur RECTANGLE COMPLET pour qu'il occupe tout l'écran.

Figure 17.1 : *Ajustement de la disposition*



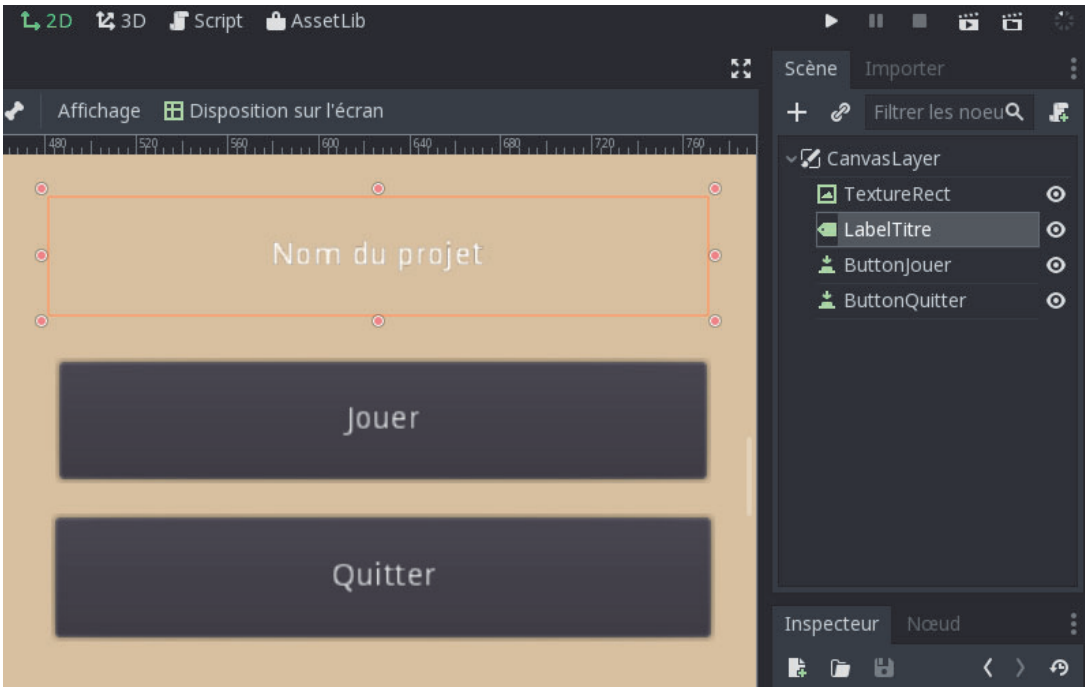
Ajoutez une image de fond via la propriété TEXTURE. Cochez également l'option EXPAND afin d'étirer l'image. Pour cet exemple, j'ai utilisé une couleur unie pour le fond.

Figure 17.2 : *Image de fond*



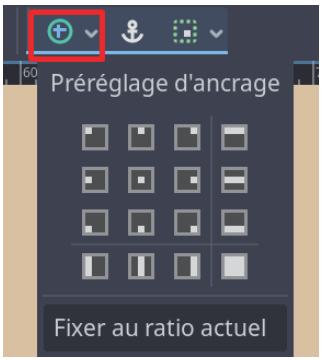
Sur cette image de fond, nous allons ajouter un nœud de type Label qui contiendra un texte (le nom de votre jeu) ainsi que deux nœuds de type Button qui seront les boutons JOUER et QUITTER. Avant de réaliser la Figure 17.3, j'ai ajouté du texte dans la propriété TEXT de chaque élément afin de mieux correspondre à ce que sera notre interface.

Figure 17.3 : *Notre menu*



Pour positionner les éléments d'interface à l'écran, vous pouvez utiliser l'outil d'ancrage se trouvant en haut de la fenêtre.

Figure 17.4 : *Ancrage d'éléments d'interface*

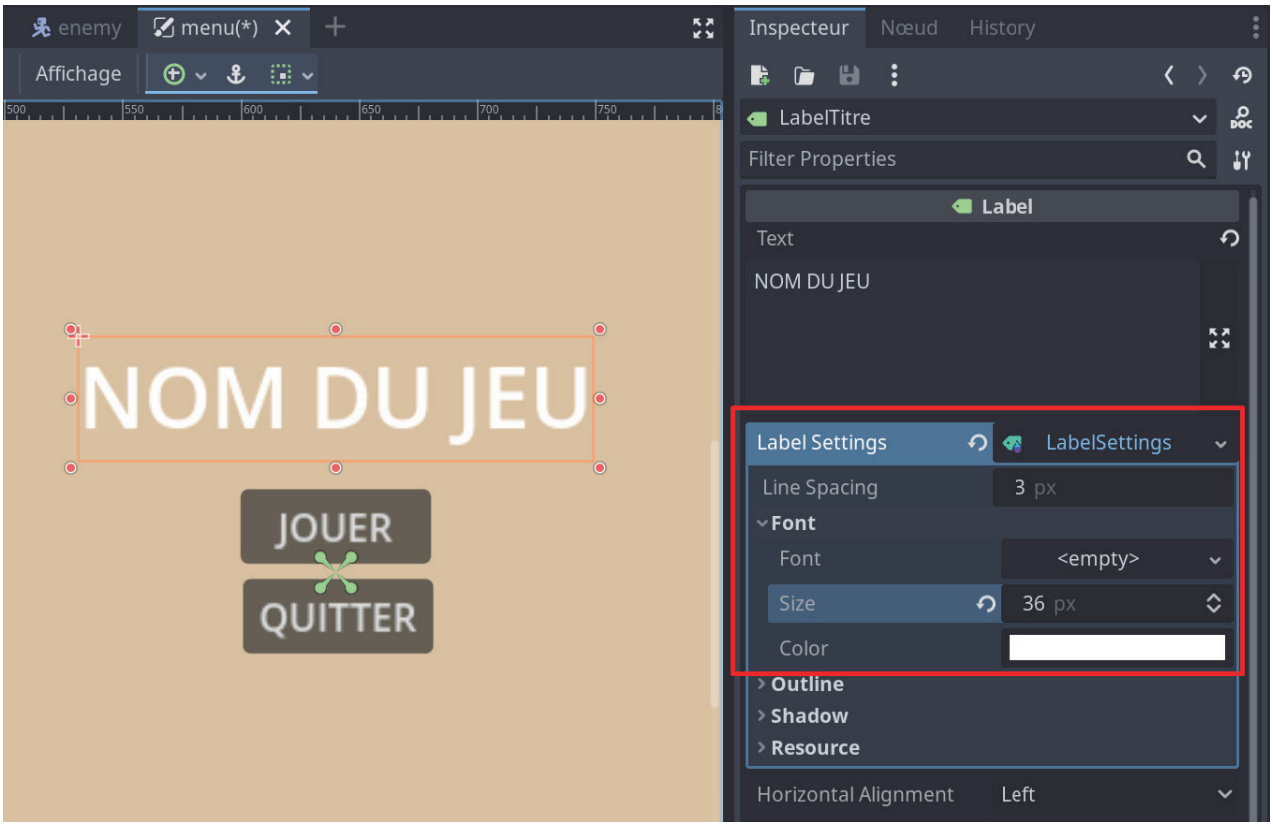


Vous pouvez ancrer les éléments dans un coin, au centre ou même étirer les éléments sur la longueur de la fenêtre. Par défaut, cet ancrage se fait par rapport à la fenêtre mais si vous construisez votre interface à l'intérieur d'un conteneur, vos éléments seront ancrés par rapport à ce conteneur. Cela vous permet de positionner les éléments avec précision. Dans mon cas, j'ai centré les éléments horizontalement et positionné manuellement les éléments verticalement. Par défaut, la police d'écriture est un peu petite. À taille réelle, le texte est peu lisible.

Figure 17.5 : *La police d'écriture par défaut*

Vous pouvez modifier la police et en créer une personnalisée. Nous avons déjà fait cette manipulation au chapitre Création de l'interface utilisateur pour l'affichage du nombre de pierres précieuses collectées. Pour chaque élément d'interface, vous pouvez ajuster la police pour l'agrandir.

Figure 17.6 : *Modification de la taille du label*



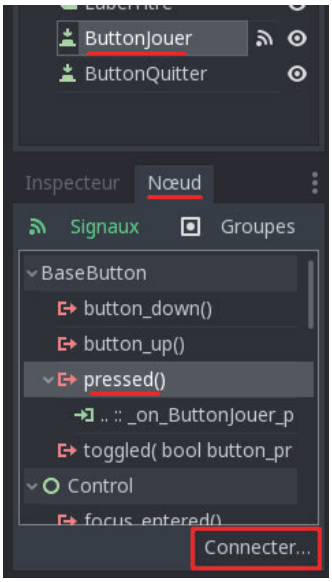
Ici il ne s'agit que d'un exemple sur le label mais vous pouvez faire de même avec les boutons. Vous pouvez également modifier la police d'écriture utilisée si vous le souhaitez.

17.2. Script du menu principal

Maintenant que nous avons notre menu, nous allons programmer les interactions avec celui-ci.

Commencez par ajouter un nouveau script vide sur votre CanvasLayer et enregistrez-le sous le nom de `Menu`. Cela nous permettra de connecter les boutons de l'interface à ce script. Retournez sur votre scène puis cliquez sur le bouton JOUER afin de le sélectionner. Sous l'onglet NŒUD, repérez l'événement `pressed()`, sélectionnez-le et cliquez sur CONNECTER pour connecter cet événement à notre script.

Figure 17.7 : *Connexion du bouton au script*



Une nouvelle fonction est ajoutée au script. Cette fonction se déclenchera lorsque le joueur cliquera sur le bouton. Pensez à déplacer cette fonction dans la classe C# si elle est créée en dehors de celle-ci. Cette fonction se chargera uniquement de lancer le jeu et donc la scène `Level1`. La méthode permettant de charger une scène est `ChangeSceneToFile`. Voici le code complet pour cette fonction :

```
// Clic sur le bouton jouer
private void _on_button_jouer_pressed()
  {
    GetTree().ChangeSceneToFile("res://Scenes/level_1.tscn");
  }
```

Faites la même chose pour le bouton QUITTER. Connectez l'événement `pressed()` au script pour générer la fonction correspondante. Celle-ci se chargera de quitter le jeu et donc de fermer l'exécutable. La méthode permettant de quitter l'application est `GetTree().Quit()`. Le code complet de votre script de menu est le suivant :

```
using Godot;
using System;

public partial class menu : CanvasLayer
{
  private void _on_button_jouer_pressed()
  {
    GetTree().ChangeSceneToFile("res://Scenes/level_1.tscn");
  }

  private void _on_button_quitter_pressed()
  {
    GetTree().Quit();
  }
}
```

Vous pouvez sauvegarder et tester le bon fonctionnement de vos deux boutons. Normalement tout devrait fonctionner comme prévu. La dernière chose que vous pouvez faire, c'est ajouter dans la fonction `_Process` du script du personnage un bout de code permettant de revenir au menu principal lorsque le joueur appuie sur la touche Échap. Le code pour faire cela est :

```
// Gestion de la touche échap < retour au menu
if (Input.IsActionJustPressed("ui_cancel"))
{
  GetTree().ChangeSceneToFile("res://Scenes/menu.tscn");
}
```

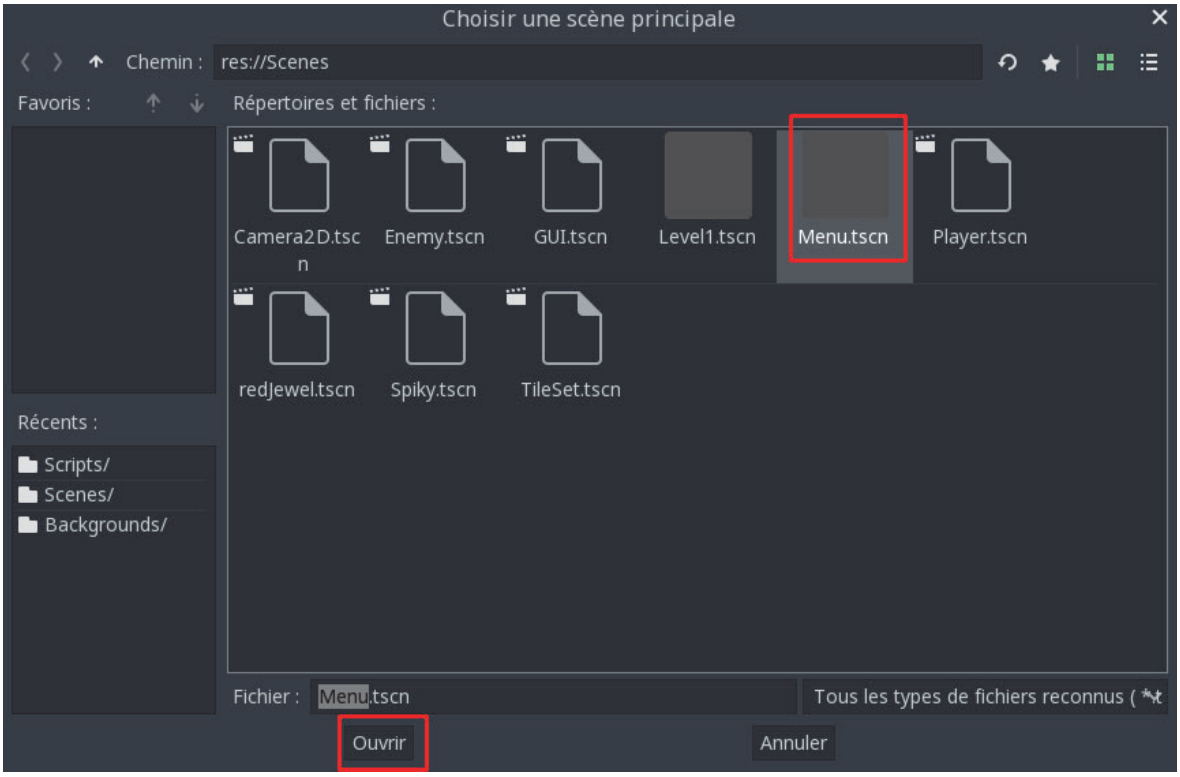
De cette façon, le joueur pourra retourner sur le menu et quitter le jeu avec le bouton approprié.

17.3. Compilation du projet

Le menu principal étant fonctionnel, il est temps de compiler notre projet pour être en mesure de le partager. La compilation consiste à générer un exécutable à partir de tous les fichiers de votre projet. Cet exécutable pourra être lancé pour démarrer le jeu vidéo.

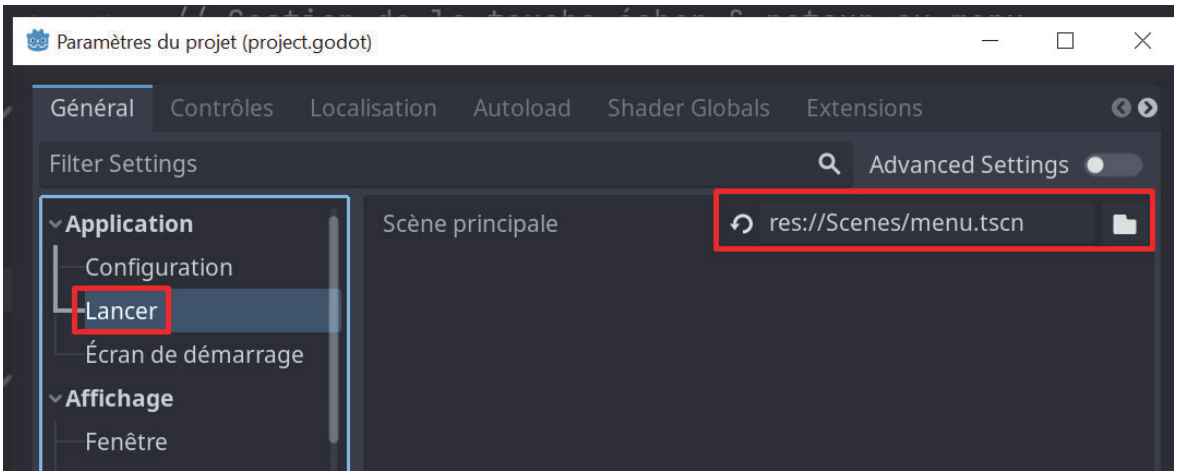
La première étape consiste à indiquer quelle est la scène principale du projet, celle qui sera lancée à l'ouverture de l'exécutable. Jusqu'ici, nous utilisions la touche F6 permettant de tester la scène en cours. Si vous appuyez sur F5, une fenêtre s'ouvrira et vous invitera à choisir une scène par défaut pour votre projet. Choisissez celle du menu principal.

Figure 17.8 : *Choix de la scène par défaut*



Ceci fait, le menu s'ouvrira systématiquement lors de l'appui sur F5 ou au lancement de l'exécutable. Cette scène par défaut peut également être spécifiée via les paramètres du projet.

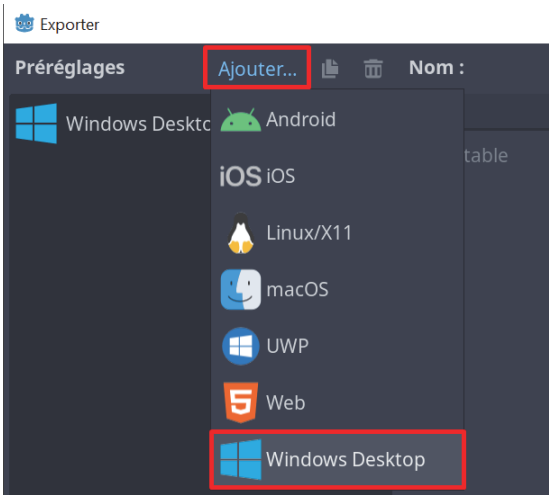
Figure 17.9 : *Paramètres du projet*



Toujours dans cette fenêtre mais sous l'onglet ÉCRAN DE DÉMARRAGE, vous avez la possibilité de définir une image qui sera affichée au lancement du jeu pendant le chargement de celui-ci. Cette image peut être votre logo par exemple.

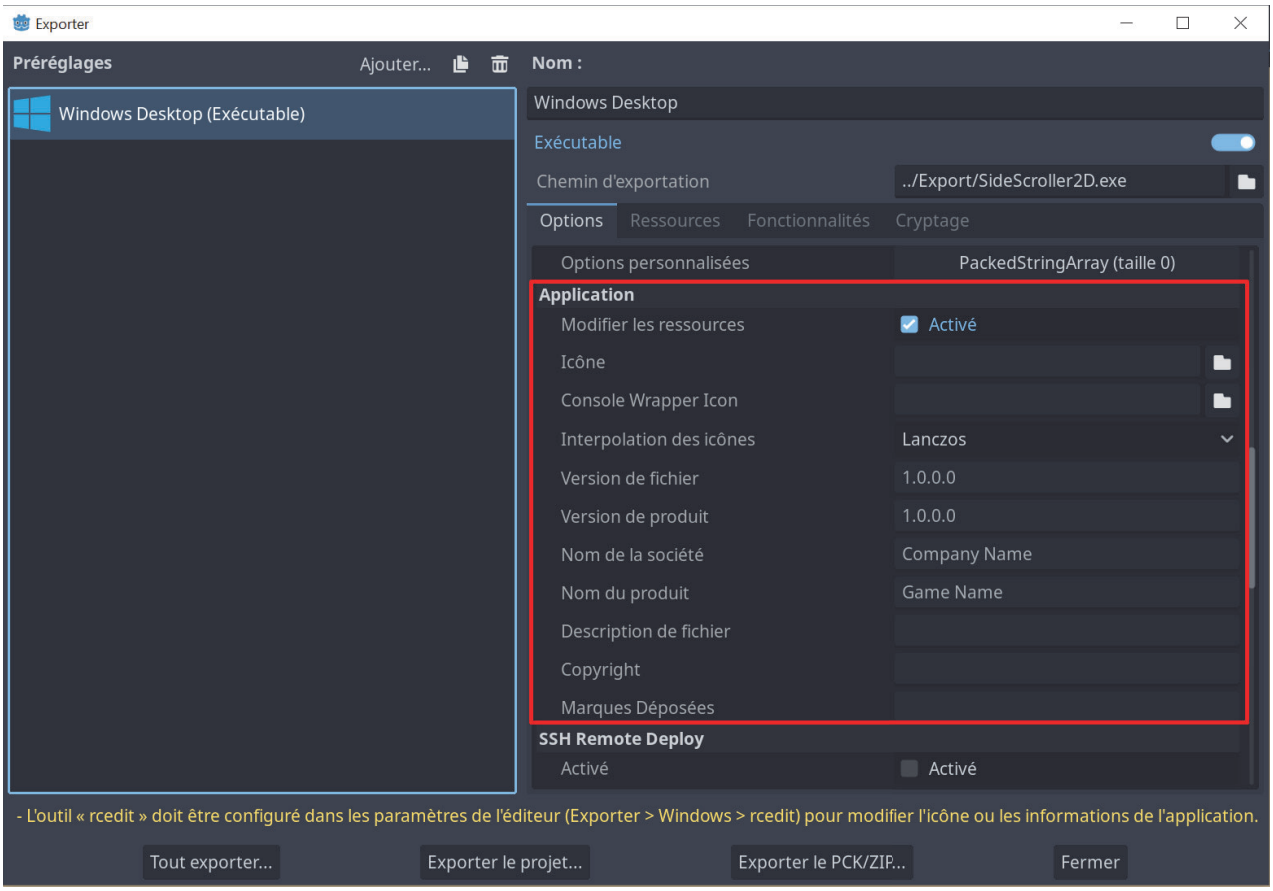
Pour exporter le projet, cliquez sur le menu PROJET/EXPORTER. Une fenêtre s'ouvre. C'est ici que sont indiquées les plateformes pour lesquelles vous pouvez réaliser un export (Windows, web, mobiles...). Pour le moment la liste est vide. Cliquez sur le bouton AJOUTER pour ajouter une plateforme. Choisissez par exemple `Windows Desktop`.

Figure 17.10 : *Ajout d'une nouvelle plateforme*



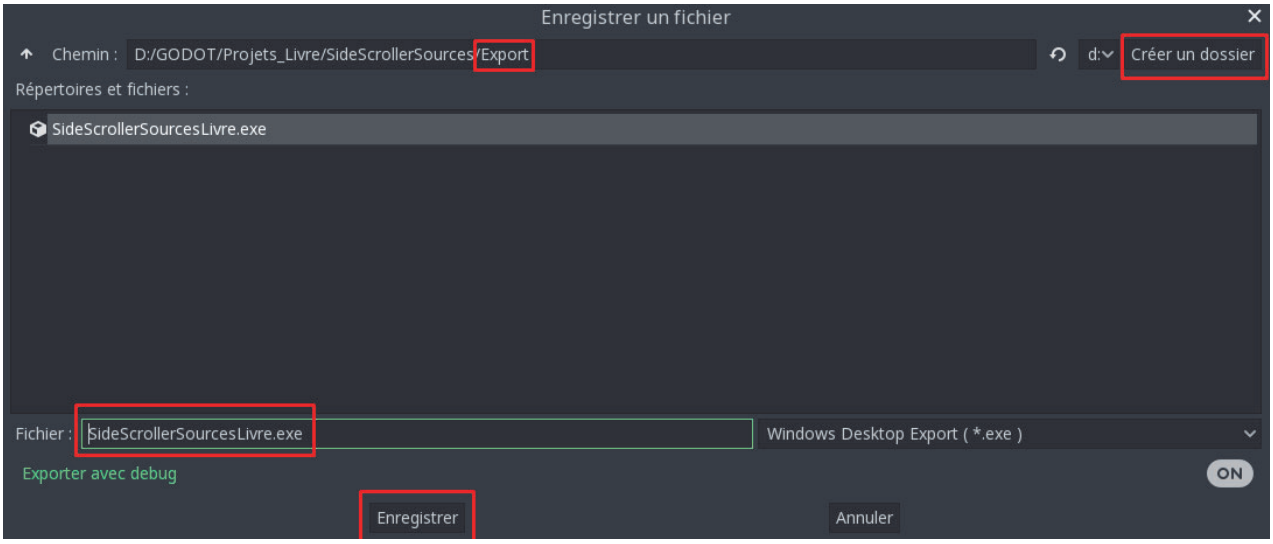
Une fois la plateforme ajoutée, vous pourrez compiler le projet. Vous avez accès à des options comme spécifier un nom, une société, un copyright ou icône pour votre projet.

Figure 17.11 : *Options de l'exécutable*



Cliquez ensuite sur EXPORTER LE PROJET puis choisissez un dossier de destination. Pour ma part, j'ai créé un dossier `Export`. Cliquez enfin sur ENREGISTRER.

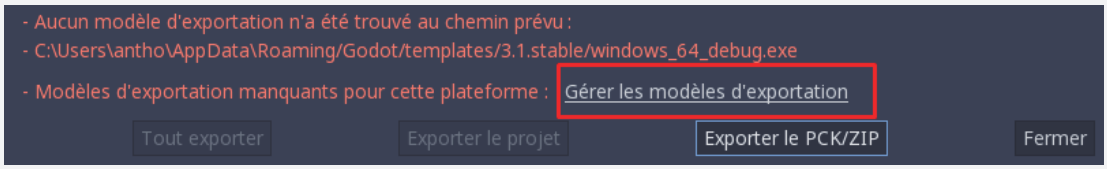
Figure 17.12 : *Exporter*



MODÈLE D'EXPORTATION MANQUANT !

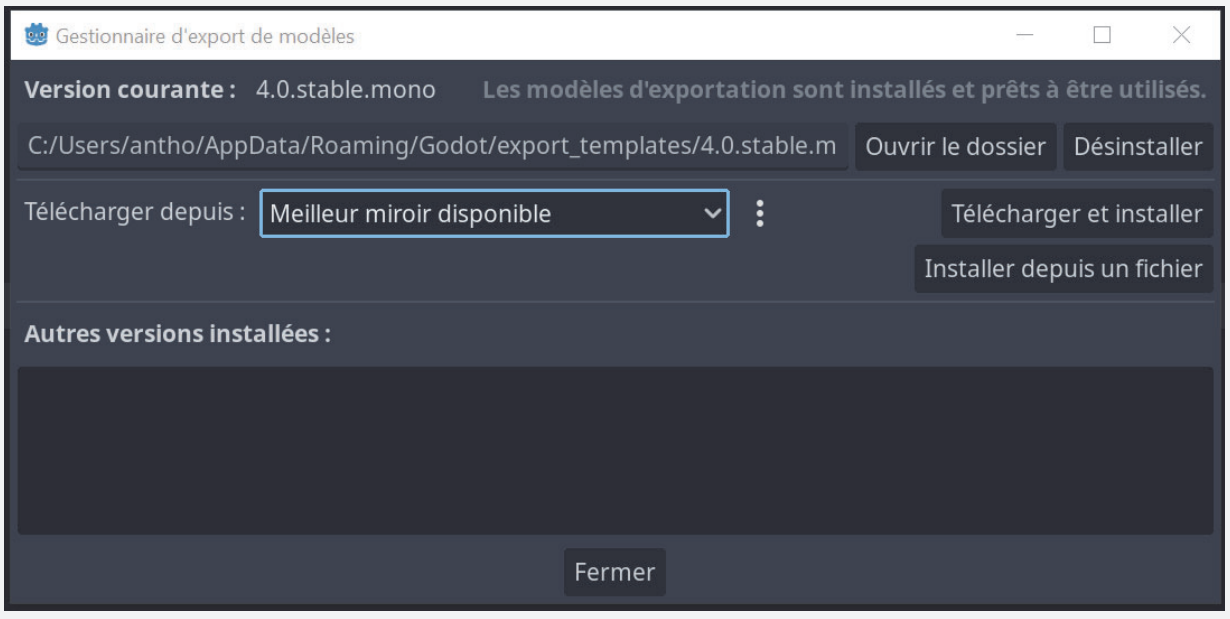
Si c'est la toute première fois que vous exportez vers une plateforme, un message d'erreur apparaîtra en bas de la fenêtre EXPORTER indiquant que le modèle d'exportation [template] est manquant, suivi d'un lien pointant vers le GESTIONNAIRE DE MODÈLES.

Figure 17.13 : *Modèle d'exportation manquant*



En cliquant sur ce lien (ou en passant par le menu EDITEUR/GÉRER LES MODÈLES D'EXPORTATION), vous accèderez à une popup comportant un bouton TÉLÉCHARGER ET INSTALLER. Procédez à l'installation. Cela installera un modèle par défaut avec le nécessaire pour générer l'exécutable de votre jeu.

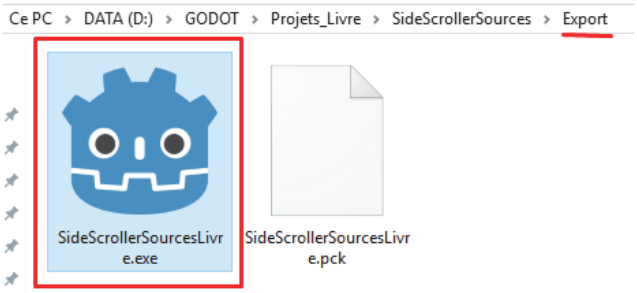
Figure 17.14 : *Ajouter un modèle d'exportation*



Vous pouvez également télécharger un modèle depuis le site de Godot et cliquer sur INSTALLER DEPUIS UN FICHIER pour installer le modèle.

Une fois terminé, l'exécutable sera généré. Vous pouvez désormais lancer le jeu avec celui-ci !

Figure 17.15 : *L'exécutable*



Nous voici parvenus à la fin de la deuxième partie. Nous avons vu ensemble les principales étapes du développement d'un jeu 2D avec Godot (mise en place des outils, création des niveaux, programmation du personnage, interactions, musiques, interface, etc.). Nous allons maintenant passer à la troisième partie dédiée à la création d'un jeu 3D. Vous le verrez, la création d'un jeu 3D diffère grandement de la création d'un jeu 2D.

Création d'un jeu 3D

Dans cette partie, nous allons créer un jeu 3D avec Godot. Dans ce jeu, le joueur aura pour mission de faire rouler une balle dans une sorte de labyrinthe 3D pour rejoindre la sortie. Même si le principe est similaire, Godot propose des composants spécifiques pour le développement de jeux 3D différents des composants 2D. Pour la partie UI (interface utilisateur), les composants sont les mêmes qu'on soit en 2D ou en 3D. Vous aurez également droit à une petite introduction préliminaire à Blender, logiciel permettant de modéliser des éléments 3D.

18

Modélisation 3D : Initiation à Blender

Comme projet d'exemple, nous allons créer un jeu de type *Marble Madness* ou *Super Monkey Ball*. Le principe est de contrôler une balle pour la faire rouler afin d'atteindre la fin du niveau tout en ramassant les bonus. Voici à quoi ressemble NeverBall, un jeu open-source dans la lignée de ceux cités précédemment :

Figure 18.1 : *Neverball*

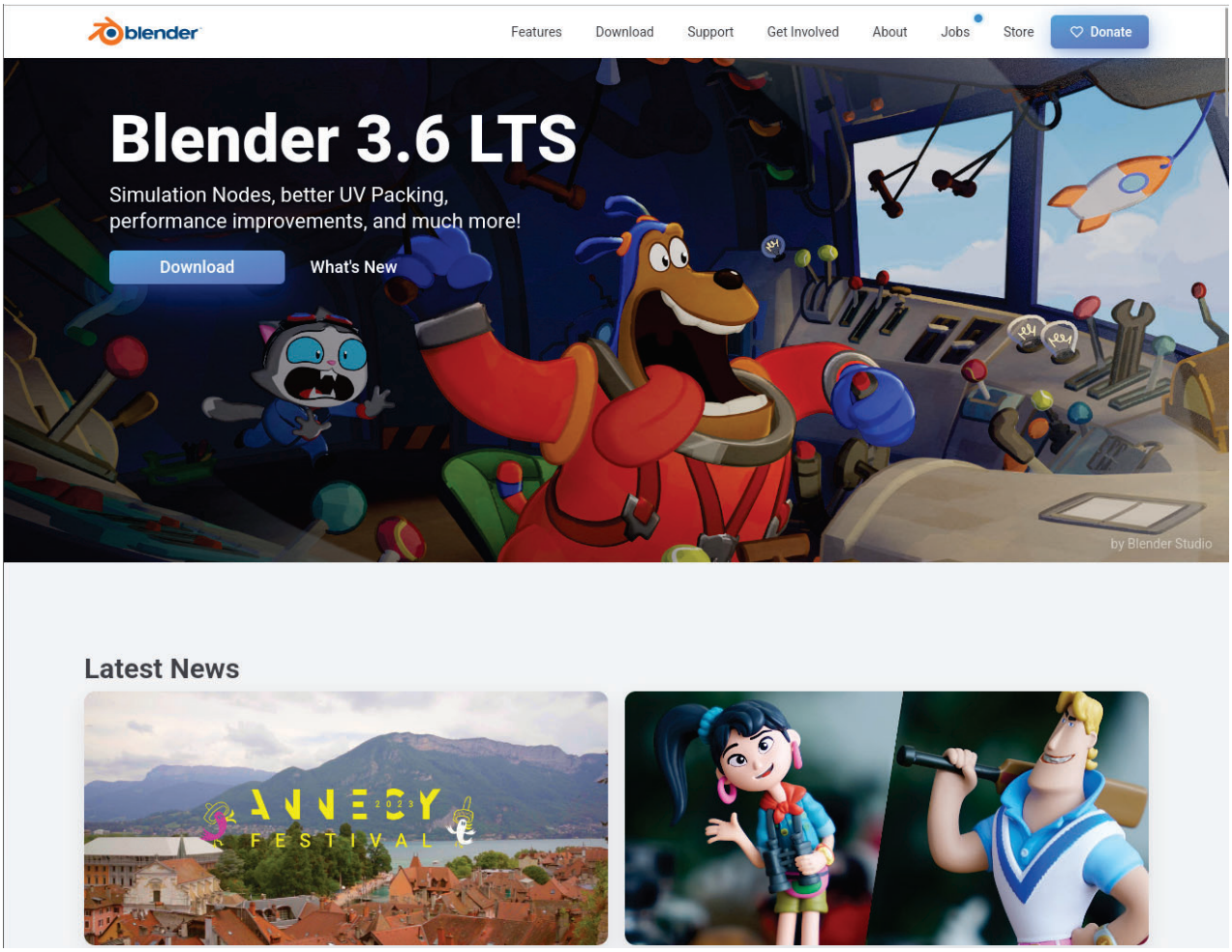
Du côté technique, ce type de jeu reste relativement simple à développer. Il faut créer la géométrie du niveau, la balle qui est le personnage, le script permettant de déplacer la balle et le système d'objets à ramasser. Une interface utilisateur simple est également importante. La dernière chose à prendre en compte côté technique est la part importante de la physique. En effet, la gravité doit affecter la balle et ses mouvements. Nous verrons tout cela au cours de cette troisième partie.

Avant de passer à la création du jeu sous Godot, nous allons commencer par modéliser les décors 3D à l'aide du logiciel Blender. Une fois que nous aurons les modèles 3D de notre jeu, nous serons en mesure de les assembler et de programmer les interactions. Ce premier chapitre sur Blender a pour but de vous présenter cet outil et son utilisation afin de partir sur de bonnes bases.

18.1. Téléchargement de Blender

Nous allons commencer par télécharger Blender. Pour cela, rendez-vous sur le site officiel du logiciel. La page DOWNLOAD vous donne accès aux différentes versions disponibles (Windows, macOS, Linux). Téléchargez la version de votre choix.

Figure 18.2 : *Page d'accueil de Blender.org*



Blender est un logiciel gratuit et open-source. Vous êtes libre de l'utiliser comme bon vous semble et d'utiliser vos créations sans restriction même pour des projets commerciaux. Une fois téléchargé, installez et lancez le logiciel.

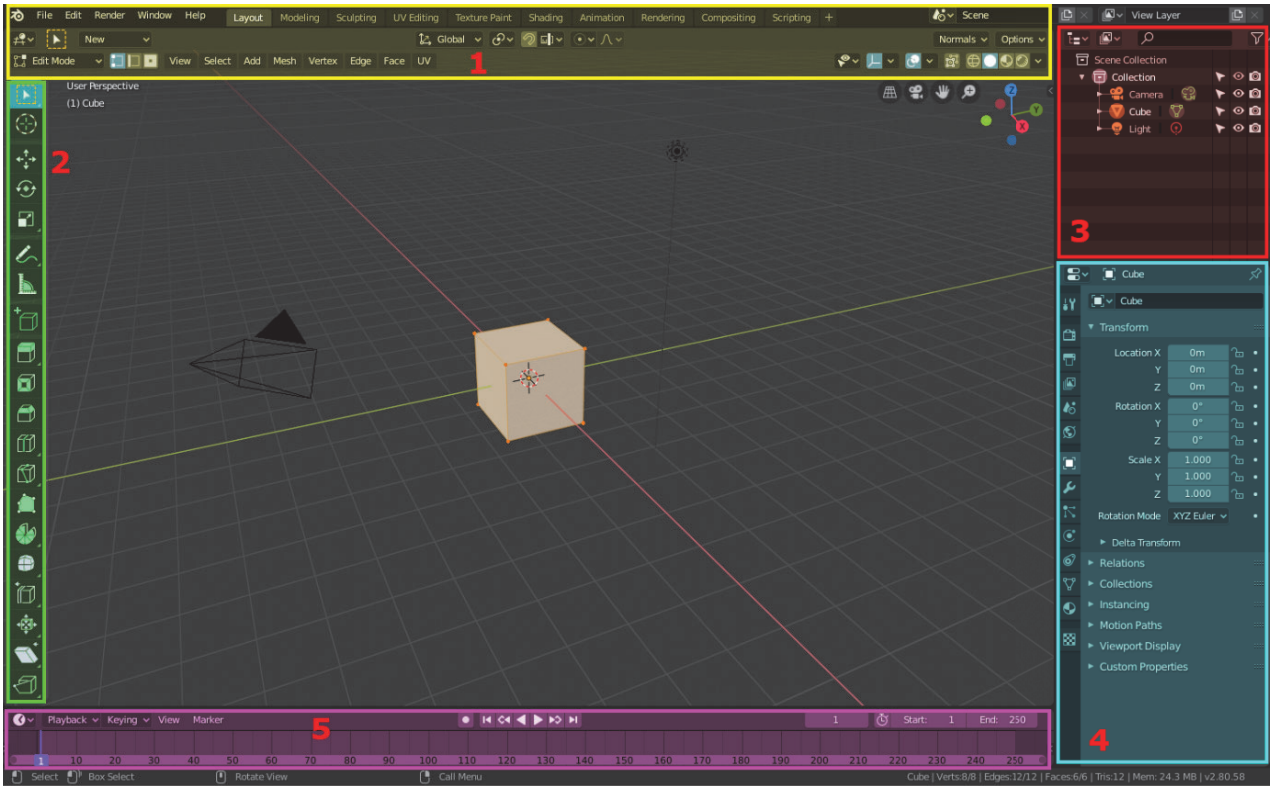
18.2. L'interface de Blender

Depuis la version 2.8 de Blender, l'interface a été totalement repensée, modernisée et surtout simplifiée. Au moment où j'écris ces lignes, Blender est en version 3.5. Son interface a peu évolué depuis cette refonte graphique. Que vous soyez sur une version 2.8, 2.9 ou 3, vous ne devriez pas vous perdre dans ce chapitre. Beaucoup de raccourcis clavier sont présents ainsi que des actions rapides pour passer très simplement d'un outil à l'autre. Comme ce livre est basé sur Godot et non Blender, je resterai sur des notions élémentaires mais suffisantes pour modéliser une forme basique.

Lorsque vous travaillez sur Blender, vous pouvez ajuster l'interface pour travailler dans un mode particulier comme le mode Objet (par défaut), le mode Édition (permettant de modifier la géométrie), le mode Peinture (permettant de texturer), etc. En fonction du mode sélectionné, l'interface peut être grandement modifiée. Pour vous présenter cette interface, j'ai fait une capture d'écran en mode Édition qui est le mode que nous utiliserons principalement.

Note *› Comme indiqué ci-dessus, Blender permet de faire beaucoup de choses (modéliser, texturer, animer, créer un rendu...). L'interface change du tout au tout en fonction du mode actif. Ici, je ne vous présenterai que l'interface standard avec laquelle nous ne faisons que de la modélisation simple.*

Figure 18.3 : *L'interface de Blender (mode Édition)*



Sur la Figure 18.3, j'ai découpé l'interface en six zones distinctes pour distinguer les principales fonctionnalités. Le centre de l'écran (qui n'a pas été détouré) est la zone de travail principale.

La zone ❶ (en jaune), où vous trouvez le menu principal permettant de créer une scène, de sauvegarder, charger, accéder à l'aide, etc. À droite du menu, vous avez des raccourcis permettant de changer rapidement de mode (objet, édition, sculpting, etc.).

La zone ❷ (en vert) vous permet d'accéder à des outils rapidement. Vous avez les outils de transformation (déplacer, tourner, agrandir l'objet) et les outils permettant de modifier la géométrie (extrusion, biseau, coupe, déformation, etc.).

La zone ❸ (en rouge) est ce que j'appelle la hiérarchie. C'est ici que vous pouvez retrouver l'arbre de votre scène. Il s'agit de la liste des objets présents dans votre scène.

La zone ❹ (en bleu clair) contient beaucoup d'outils qui peuvent être appliqués à la scène ou à l'objet. Nous utiliserons particulièrement les *modifiers* qui se trouvent ici. Vous aurez aussi accès aux matériaux permettant de colorer l'objet ou aux propriétés de transformation.

La zone ❺ (en violet) est l'outil d'animation qui est utilisé pour travailler avec les animations d'objets et de personnages.

La zone principale, au centre de l'écran, est la zone sur laquelle vous travaillerez majoritairement. C'est là que vous effectuerez vos modélisations 3D ou la création de textures pour vos objets 3D.

Comme je vous l'ai dit, Blender dispose de beaucoup de modes de travail et chaque mode a sa propre interface. Ici nous n'avons que survolé celle-ci mais nous verrons plus en détail les outils lors de la création de nos modèles 3D.

18.3. Les raccourcis et outils indispensables

La modélisation 3D est un métier complexe qui demande des années de pratique. Ce livre n'a pas pour but de faire de vous un professionnel de la modélisation 3D mais simplement de vous enseigner quelques techniques qui vous permettront d'apprendre rapidement et facilement à modéliser des objets 3D basiques. En effet, même si la modélisation 3D est un vrai métier, il existe quelques méthodes très simples à apprendre qui vous permettront de réaliser 75% de ce qui sortira de votre imagination. Une dizaine de techniques élémentaires sont largement suffisantes pour réaliser des décors comme celui de la Figure 18.4.

Figure 18.4 : Exemple de décor simple fait avec Blender

Note *› Cette image illustre un kit d'éléments 3D* lowpoly *que je propose en téléchargement sur mon site internet. Ce kit contient des dizaines de modèles 3D que vous pouvez utiliser pour créer des décors et des plateformes dans vos jeux.*

Le zip contient le projet Godot avec les ressources directement importées. Vous pouvez utiliser le projet complet ou simplement extraire les modèles 3D. Vous retrouverez également le fichier source Blender si vous souhaitez éditer les modèles 3D et les réexporter.

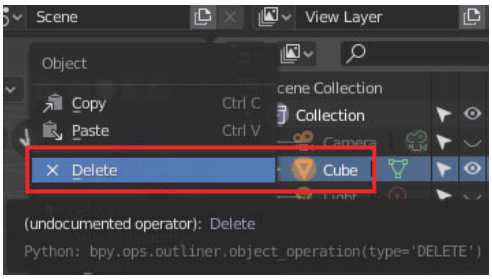
Dans ce chapitre, je vais vous présenter les raccourcis clavier et les outils que vous devez impérativement mémoriser pour mener à bien vos projets. Bien qu'il existe de nombreux modèles 3D téléchargeables en ligne, il est important d'apprendre à se débrouiller pour rendre vos jeux réellement uniques.

Une dernière chose : Blender fonctionne énormément avec les raccourcis clavier. En fait, chaque manipulation est associée à un raccourci, ce qui fait beaucoup de choses à retenir mais avec de la pratique vous y arriverez et vous serez beaucoup plus rapide. Je vous donnerai les raccourcis associés à toutes les manipulations que je vais vous présenter ici.

Création rapide

Par défaut, lorsque vous lancez Blender, un cube se trouve sur la scène. Pour le supprimer, faites un clic droit sur Cube via la hiérarchie puis DELETE (supprimer).

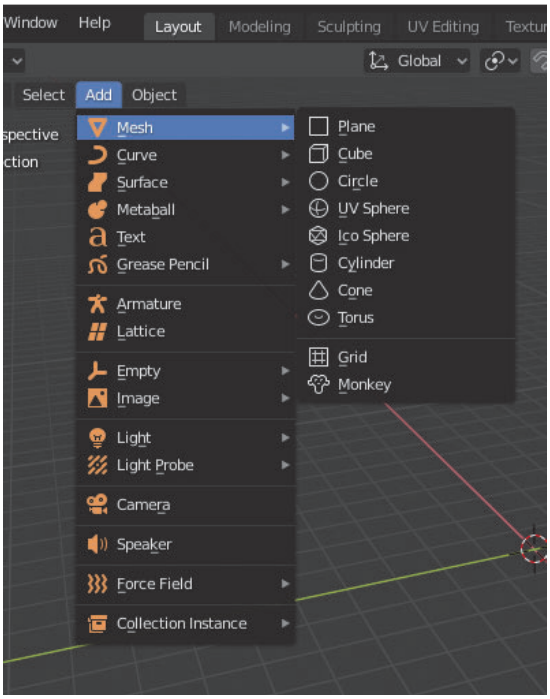
Figure 18.5 : *Supprimer un objet*



Le raccourci clavier pour effectuer cela est X puis DELETE. Avant d'appuyer sur X, pensez à sélectionner le cube dans la zone principale avec un clic gauche (ou clic droit si ancienne version de Blender).

Maintenant que la scène est vide, voyons comment ajouter un objet. Pour ajouter un objet comme un cube, une sphère ou un autre type de primitive, cliquez sur ADD/MESH/CUBE.

Figure 18.6 : *Ajout d'une primitive*

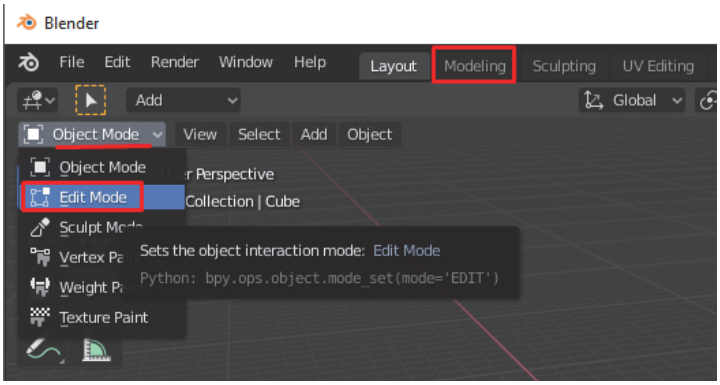


Le raccourci clavier associé à cette action est Maj+A. Pour la plupart des modélisations, nous commencerons par créer un cube. Vous pouvez donc ajouter un cube à la scène.

Changer de mode

Dans Blender, par défaut, nous nous trouvons en mode Objet permettant de travailler l'objet dans sa globalité. Le mode que nous utiliserons régulièrement est le mode Édition [Edit Mode]. Pour passer dans ce mode, cliquez sur l'onglet MODELING ou sélectionnez EDIT MODE dans le menu déroulant à gauche portant le label du mode en cours (voir Figure 18.7).

Figure 18.7 : *Passer en mode Édition*

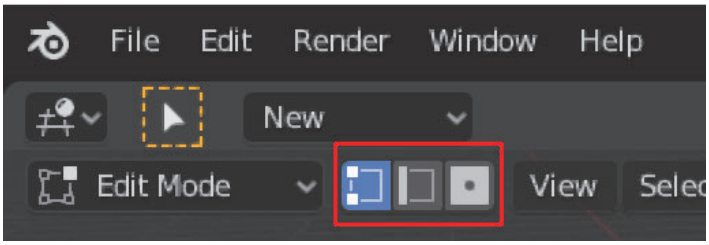


Le raccourci clavier pour basculer d'un mode à l'autre est la touche Tabulation. Essayez de vous souvenir des combinaisons de touches, il est toujours plus pratique de passer par ces raccourcis. En mode Édition, vous allez pouvoir travailler la géométrie de l'objet (les points, arêtes et faces).

Points, arêtes, faces

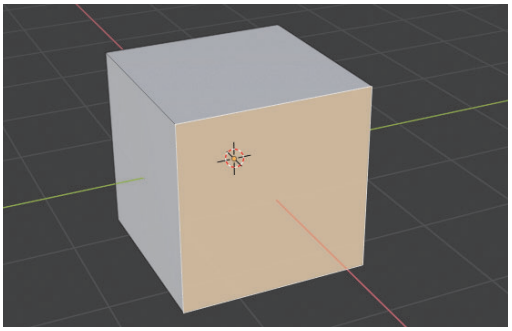
Pour éditer un objet, choisissez au préalable le type d'éléments sur lequel vous souhaitez intervenir : points ou arêtes ou faces en cliquant sur le bouton adéquat.

Figure 18.8 : *Boutons d'édition des points, arêtes et faces*



Si par exemple, je me mets en mode Faces , je peux éditer une face du cube.

Figure 18.9 : *Mode Faces sur le cube*

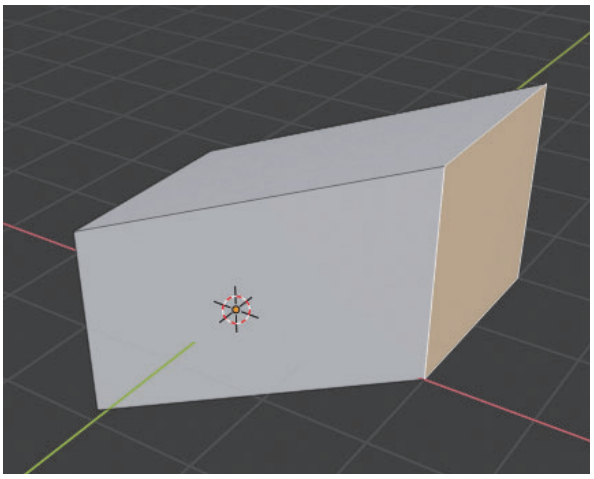


En maintenant la touche Maj enfoncée, vous pourrez effectuer une sélection multiple.

Les transformations

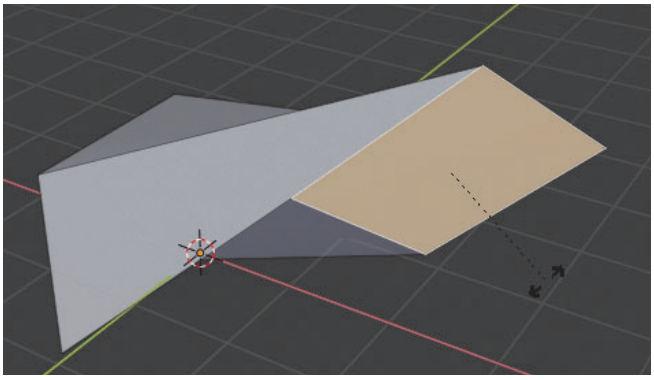
Ce que j'appelle les transformations, c'est le déplacement, la rotation ou le redimensionnement. Avec une face du cube sélectionnée, appuyez sur la touche G puis déplacez votre souris pour effectuer un déplacement.

Figure 18.10 : *Déplacer*



Avec une face sélectionnée, appuyez sur R pour appliquer une rotation en déplaçant la souris.

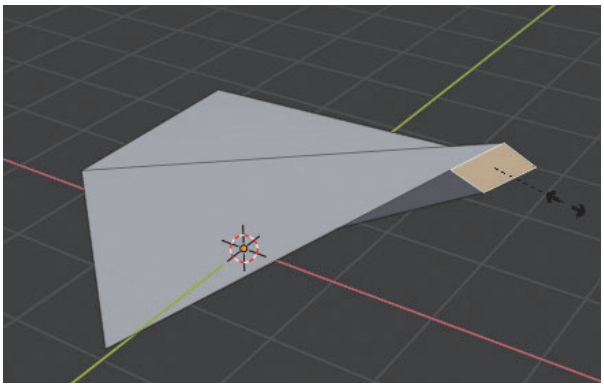
Figure 18.11 : *Rotation*



Note *› Comme vous pouvez le constater, l'action (déplacement, rotation, etc.) s'effectue uniquement sur la ou les faces sélectionnées. Le modèle 3D est déformé en fonction. Si vous sélectionnez toutes les faces, l'objet tournera entièrement. En mode Objet (le mode par défaut, ici nous sommes en mode Édition), la transformation est appliquée sur l'objet en entier.*

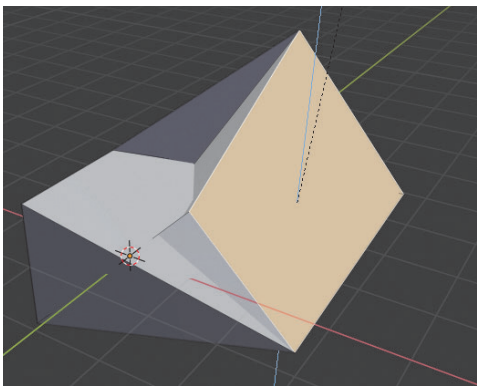
Enfin, la touche S permet de redimensionner :

Figure 18.12 : *Scale*



Si vous voulez appliquer une transformation sur un axe précis (X, Y ou Z), appuyez sur X, Y ou Z. Votre transformation s'appliquera sur l'axe choisi.

Figure 18.13 : *Scale en Z uniquement*

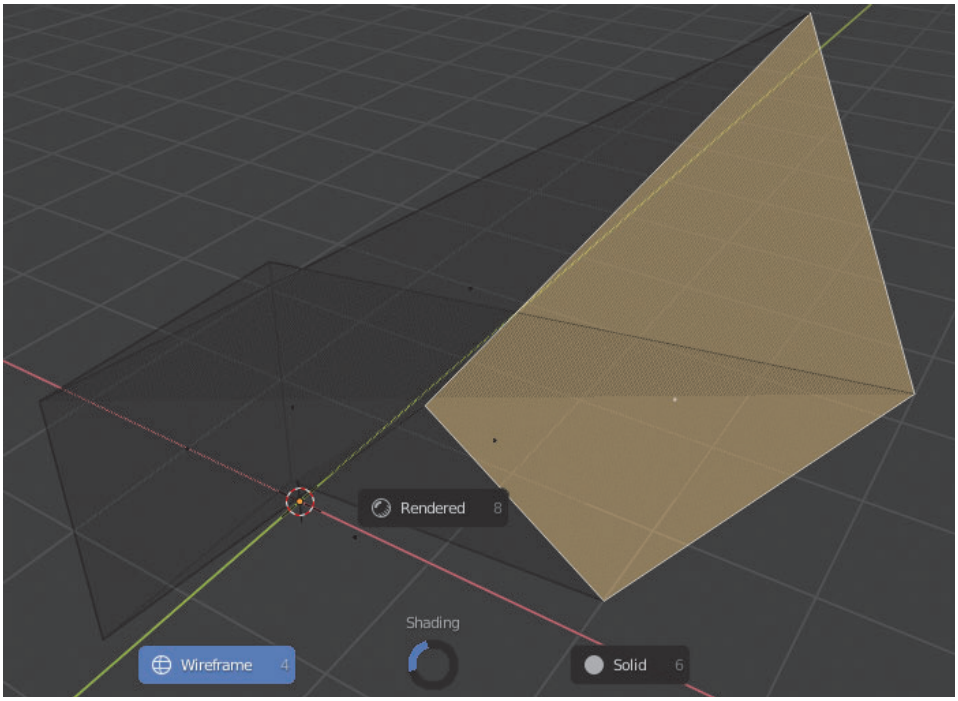


Si vous voulez appliquer la transformation sur tous les axes sauf un axe spécifique, utilisez Maj + l'axe. Exemple : Maj+Z pour transformer en X et Y.

Mode Transparence

Vous pouvez passer en transparence pour voir au travers des faces. Pour cela, utilisez la touche Z. Choisissez WIREFRAME sur la nouvelle version de Blender.

Figure 18.14 : *Wireframe*

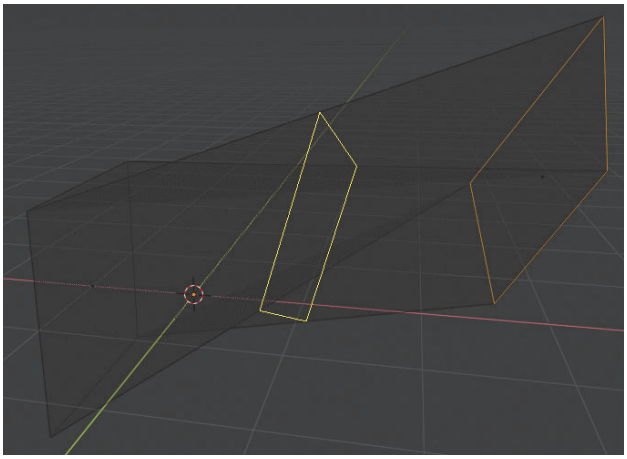


En mode Transparence, vous aurez plus de facilités à sélectionner des points ou faces. Vous pourrez aussi sélectionner toutes les faces même cachées avec les outils de sélection que nous allons voir.

Couper le modèle 3D

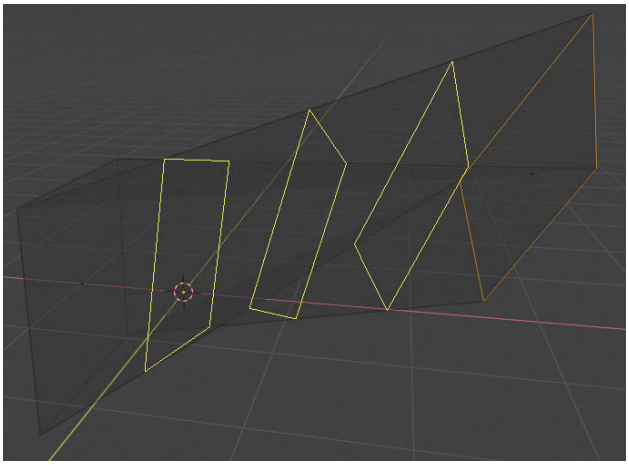
Il est possible de découper un modèle 3D notamment pour ajouter des arêtes et travailler plus précisément le maillage. Utilisez Ctrl+R pour réaliser une coupe circulaire.

Figure 18.15 : *Coupe*



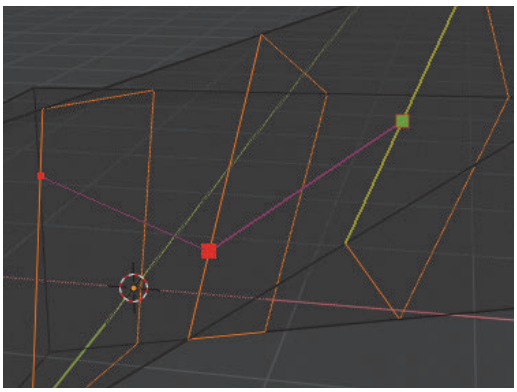
En mode Coupe, utilisez la molette de la souris pour effectuer une coupe multiple.

Figure 18.16 : *Coupe multiple*



Cliquez pour valider la coupe du modèle. Il existe d'autres outils de coupe comme le couteau (raccourci K) permettant de tracer la coupe.

Figure 18.17 : *Outil Couteau*



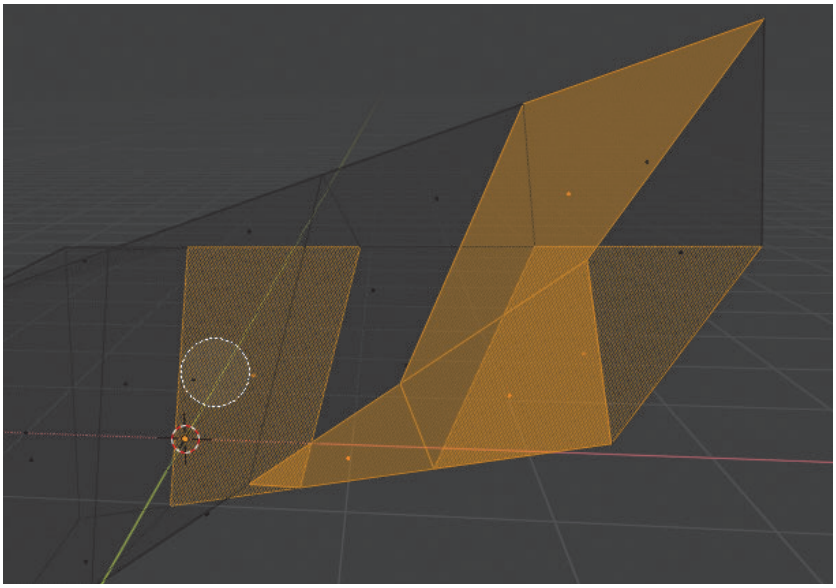
Les outils de coupe sont accessibles via les icônes à gauche de l'interface :

Figure 18.18 : *Outils de coupe*

Outils de sélection

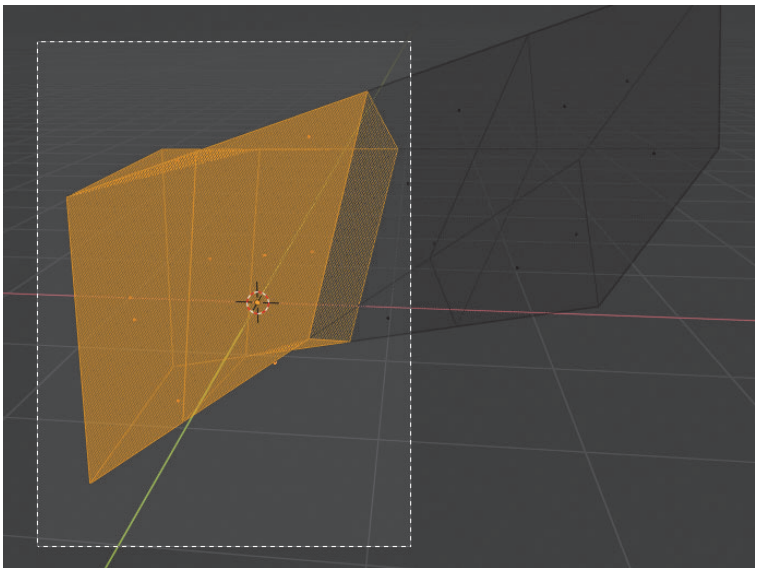
Vous pouvez sélectionner des faces via plusieurs techniques. La touche A permet de tout sélectionner/désélectionner. La touche C permet de sélectionner des faces, points ou arêtes avec un pinceau.

Figure 18.19 : *Pinceau de sélection*



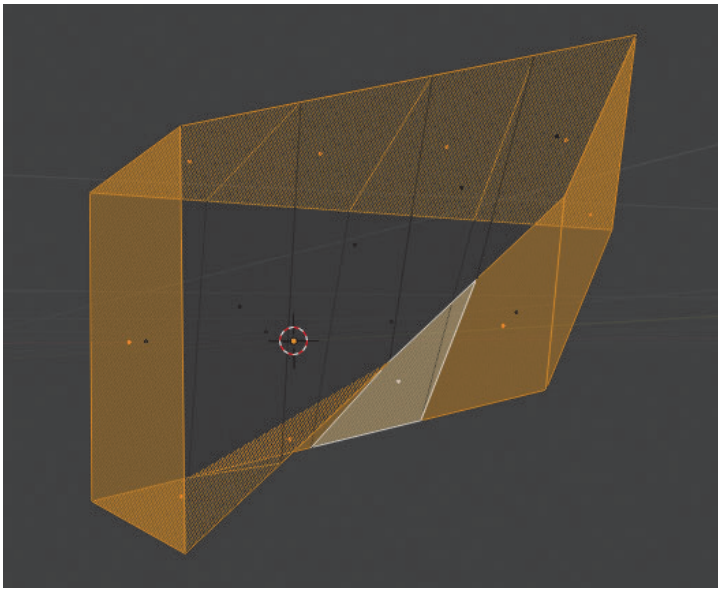
La touche B permet de sélectionner tout ce qui se trouve dans une boîte de sélection.

Figure 18.20 : *Boîte de sélection*



Enfin, la combinaison Alt+Clic (ou Alt+Clic droit) permet de faire une boucle de sélection.

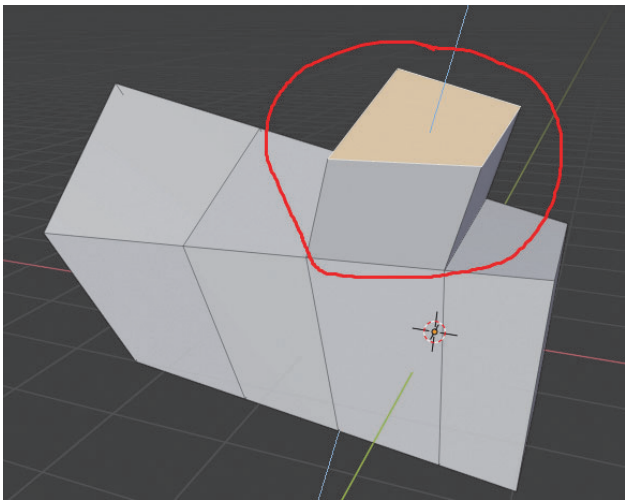
Figure 18.21 : *Sélection en boucle*



Extrusion

L'extrusion est l'outil le plus utilisé pour modéliser et ajouter des polygones. Le principe est simple : vous sélectionnez une face puis vous appuyez sur E pour réaliser l'extrusion et vous déplacez votre souris. Cela permet de faire sortir des polygones de votre sélection.

Figure 18.22 : *Extrusion*

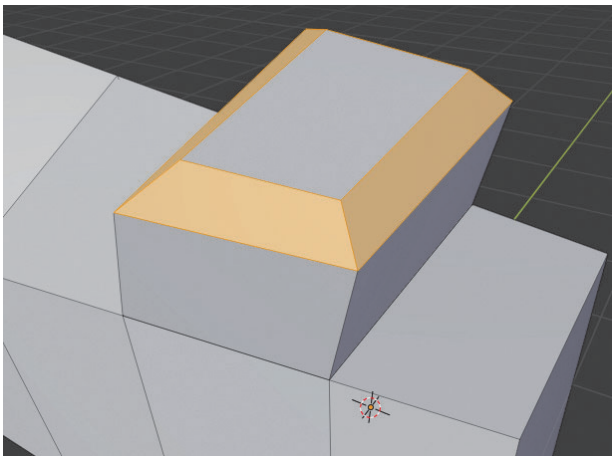


Comme pour beaucoup d'outils, vous pouvez appuyer sur X, Y ou Z pour appliquer l'extrusion sur l'axe donné. Ou Maj+l'axe pour appliquer sur tout sauf l'axe.

Biseau

Vous serez également souvent emmené à effectuer un biseau avec le raccourci Ctrl+B.

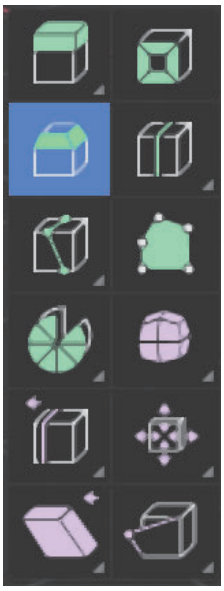
Figure 18.23 : *Biseautage*



Récapitulatif

Vous retrouverez la liste des outils courants dans le menu se trouvant sur la gauche.

Figure 18.24 : *Outils courants*



Pour vous aider à vous souvenir des raccourcis clavier, je vous ai fait un mémo Blender que vous pouvez télécharger sur le site des éditions D-BookeR. Avec ces quelques outils, vous serez en mesure de modéliser beaucoup de choses dont les éléments constituant les niveaux de notre jeu 3D. Ces quelques techniques permettent par exemple de créer les éléments 3D du kit de la Figure 18.4.

Note *› Si vous souhaitez d'autres exemples pour vous entraîner à manipuler Blender, j'ai écrit un livre dédié, également publié aux éditions D-BookeR :* Création d'assets 3D pour le jeu vidéo avec Blender.

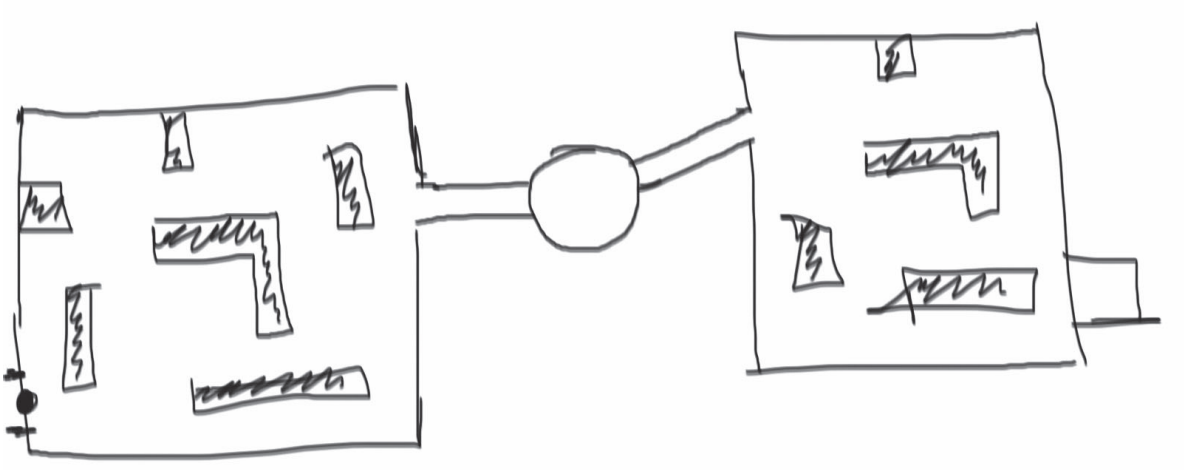
Au chapitre suivant, nous verrons comment modéliser le premier niveau de notre jeu 3D.

19

Modélisation 3D du niveau de notre jeu

Maintenant que vous avez quelques notions des techniques de modélisation avec Blender, nous allons passer à la modélisation du premier niveau de notre jeu. Celui-ci sera simple, il sera constitué de deux zones reliées par un pont comme dessiné sur le croquis ci-dessous.

Figure 19.1 : *Croquis du niveau 1*



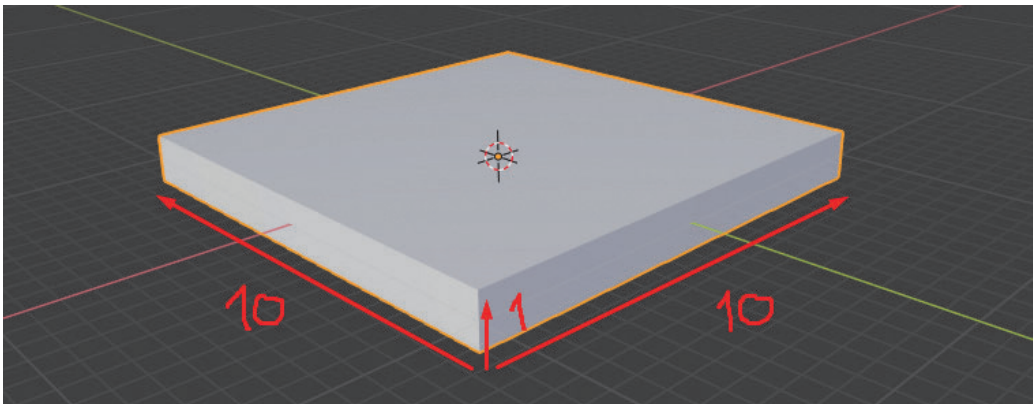
Les petits rectangles représentent des sortes d'obstacles qu'il faudra contourner. La difficulté de base sera de ne pas tomber dans le vide. Vous pourrez ensuite ajouter de la difficulté par d'autres biais.

19.1. Création des zones

Sous Blender, créez un nouveau projet et conservez le cube créé par défaut ; nous allons modéliser le niveau à partir de celui-ci. Supprimez la caméra et la lumière se trouvant par défaut sur la scène, nous n'en n'aurons pas besoin.

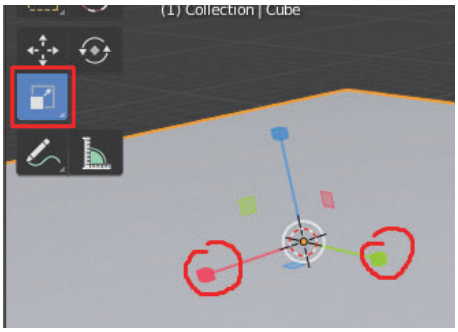
Tout en restant en mode Objet (mode par défaut), appuyez sur S pour redimensionner le cube, puis sur Maj+Z pour ne pas prendre en compte l'axe Z. Enfin, appuyez sur les chiffres 1 et 0 pour écrire "10" afin que la taille du cube soit de 10 unités.

Figure 19.2 : *Redimensionner le cube*



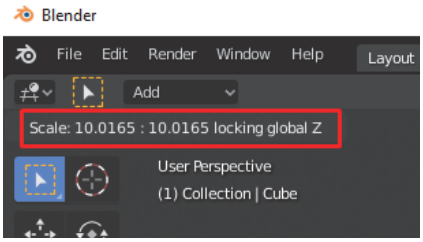
Si jamais vous souhaitez redimensionner manuellement le cube sans utiliser les raccourcis clavier, vous pouvez utiliser l'outil Scale et les poignées.

Figure 19.3 : *Redimensionner*



Pour vérifier la taille du cube (et travailler avec précision), regardez en haut à gauche de votre écran. La taille s'y affiche. Si vous voulez travailler avec des valeurs rondes et précises, maintenez la touche Ctrl enfoncée pendant que vous redimensionnez.

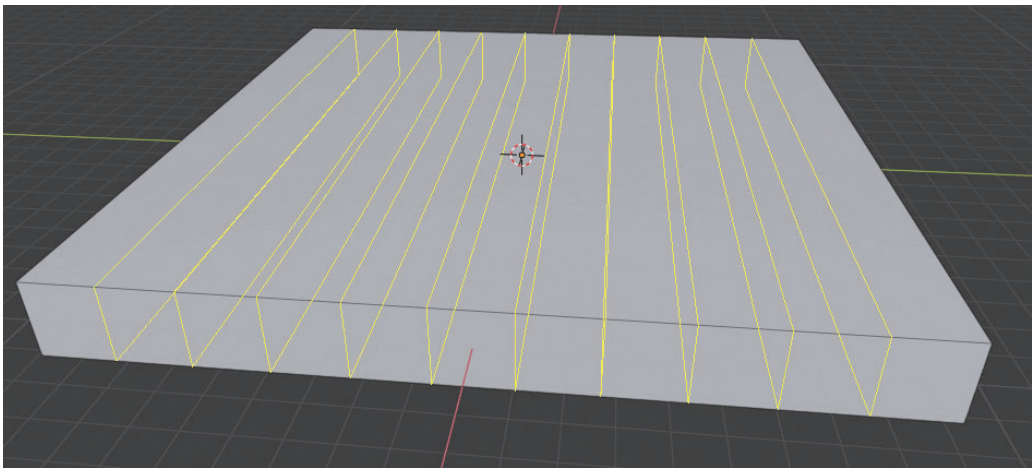
Figure 19.4 : *Taille de l'objet*



Nous avons donc la base de notre première plateforme. Passez en mode Édition en utilisant le menu adéquat ou en utilisant la touche Maj. Afin de créer la structure de notre

niveau, nous allons découper notre cube en cases. Il faut que celui-ci soit constitué de 10 × 10 cases. Faites un Ctrl+R pour activer la coupe en boucle. En mode Coupe, utilisez la molette de votre souris pour faire apparaître 10 boucles (oui, il vous faut absolument une souris pour modéliser, oubliez le TouchPad !).

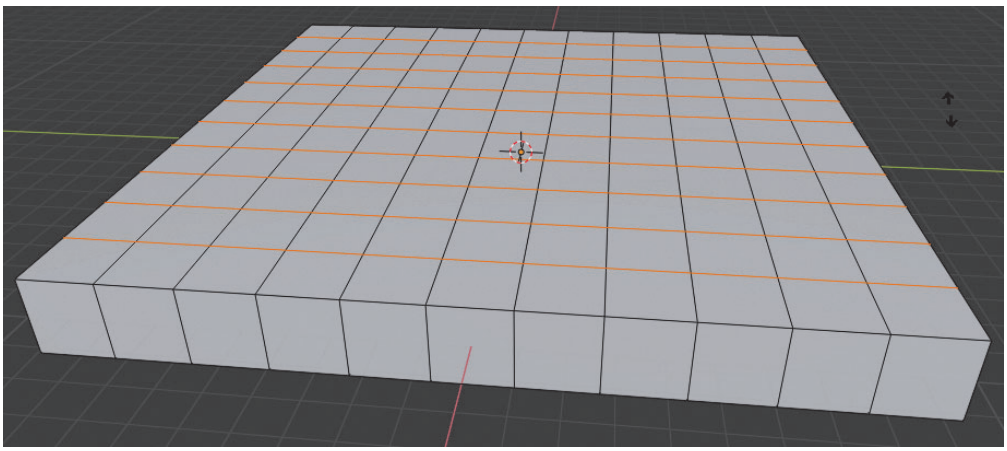
Figure 19.5 : *Couper en 10*



Cliquez pour valider le nombre de coupes et cliquez à nouveau pour valider la position des coupes. Attention à ne pas bouger la souris sinon vos coupes seront décalées.

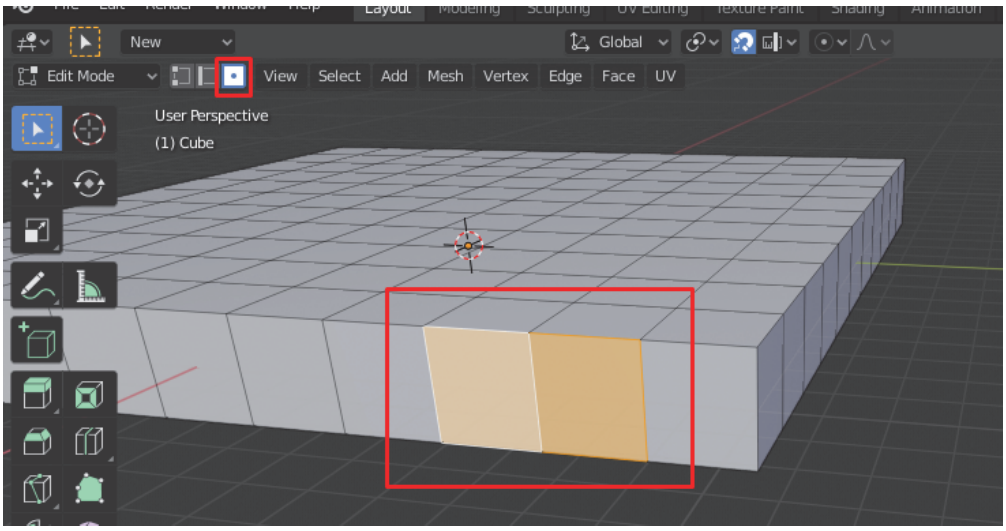
Coupez de nouveau perpendiculairement. Pour cela, répétez exactement la même manipulation mais en plaçant le curseur de la souris sur les côtés du cube pour que la coupe soit faite dans l'autre sens, puis validez pour obtenir le résultat de la Figure 19.6.

Figure 19.6 : *Coupe de 10 par 10*



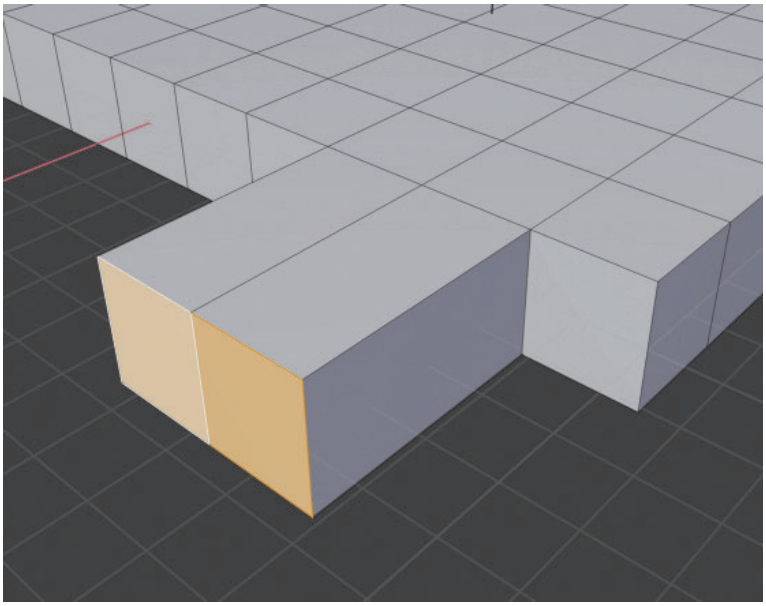
Maintenant que nous avons la structure de base, nous allons créer une petite zone qui sera le point de départ du niveau. Pour cela, sélectionnez deux cases sur un coin de la forme.

Figure 19.7 : *Sélection de deux cases*



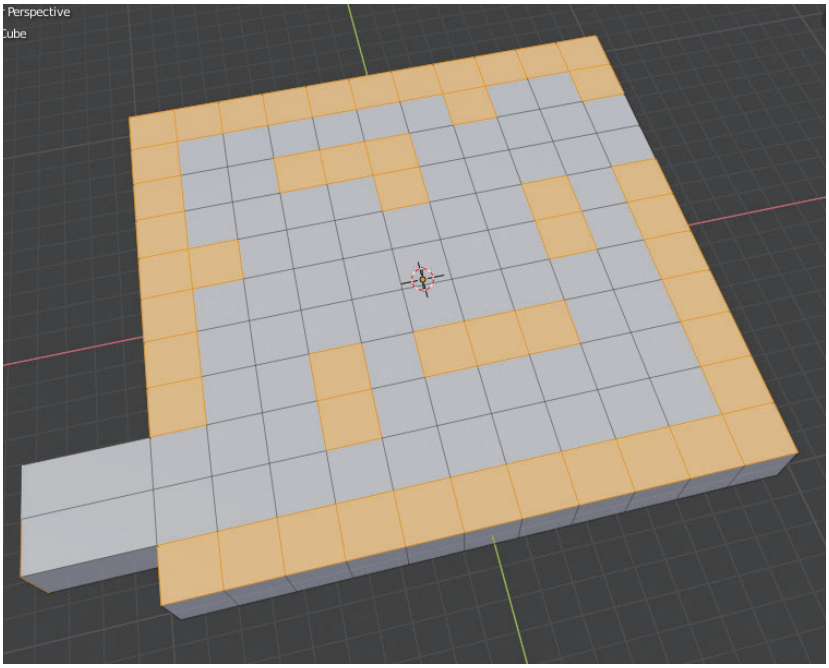
Vous devez être en mode Faces et maintenir la touche Maj enfoncée quand vous cliquez pour faire cette sélection multiple. Appuyez ensuite sur E pour effectuer une extrusion.

Figure 19.8 : *Extrusion des deux cases*



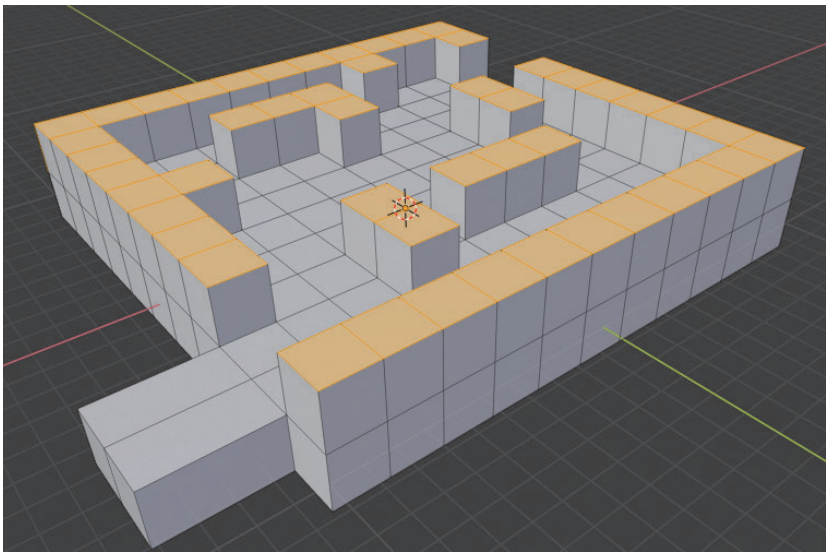
Nous allons maintenant créer les murs de la plateforme. Pour cela, sélectionnez tous les cubes qui forment le contour. Pensez à laisser deux cases pour créer un couloir de sortie de la plateforme. Sélectionnez également des cubes sur la plateforme pour créer une sorte de labyrinthe. Aidez-vous du croquis pour faire cette sélection. La Figure 19.9 vous montre un exemple de sélection.

Figure 19.9 : *Sélection des murs*



Effectuez ensuite une extrusion vers le haut de deux unités :

Figure 19.10 : *Extrusion des murs*

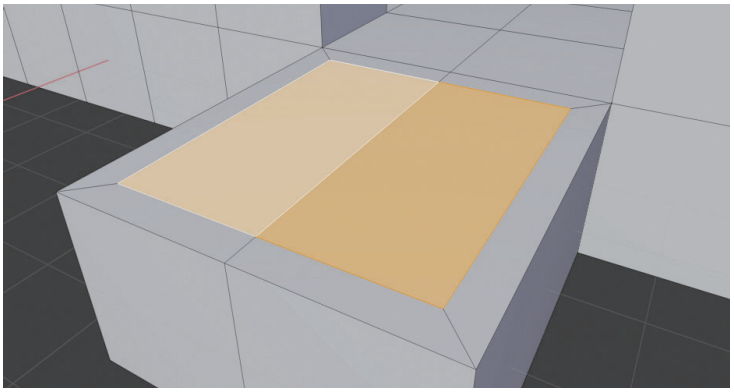


Vous pouvez bien sûr adapter la forme si vous le souhaitez. Quoi qu'il en soit, il s'agit du premier niveau du jeu, vous pourrez par la suite créer des dizaines de plateformes différentes pour varier les niveaux et la difficulté. Pour le premier niveau, je souhaite mettre des murs autour de la forme pour que le joueur ne puisse pas tomber. Il lui faut un

petit temps d'adaptation pour apprendre à manier la balle et à se déplacer, la difficulté doit arriver après.

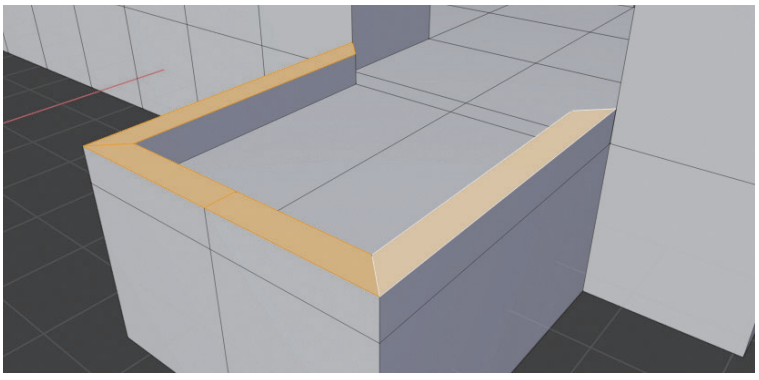
Revenez au niveau de la petite avancée représentant le début du niveau. Sélectionnez les deux polygones (les deux faces) du dessus. Appuyez sur la touche I pour incruster des faces. Déplacez la souris pour faire cette insertion. Essayez d'obtenir quelque chose ressemblant à la Figure 19.11.

Figure 19.11 : *Insertion de faces*



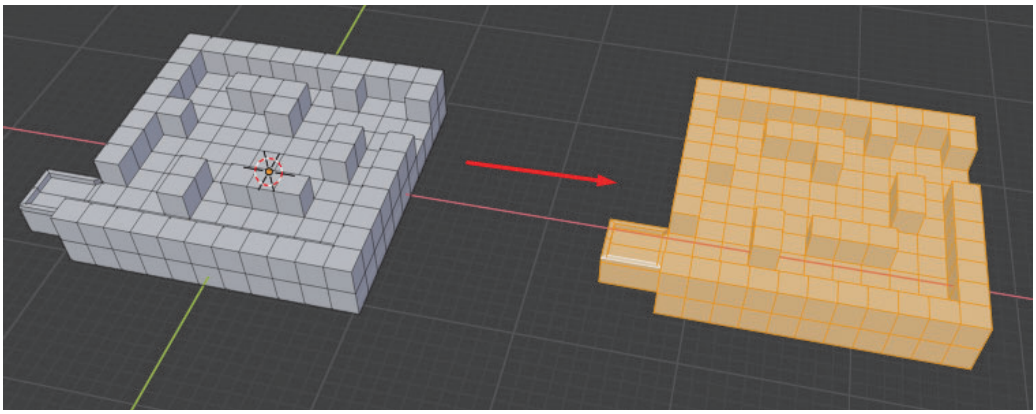
Cela va nous permettre d'effectuer une petite extrusion pour créer un rebord à notre plateforme et éviter que le joueur ne tombe au lancement du jeu. Vous pouvez maintenir la touche Ctrl enfoncée pour travailler avec un peu plus de précision.

Figure 19.12 : *Petite extrusion sur la plateforme*



Nous allons maintenant dupliquer cette plateforme. Appuyez sur la touche A pour sélectionner l'intégralité de la plateforme, puis Maj+D pour la dupliquer. Placez la nouvelle plateforme sur le côté avec la souris.

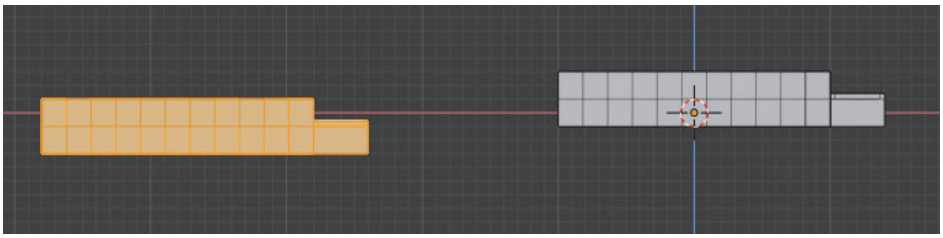
Figure 19.13 : Dupliquer



19.2. Alignement des deux zones

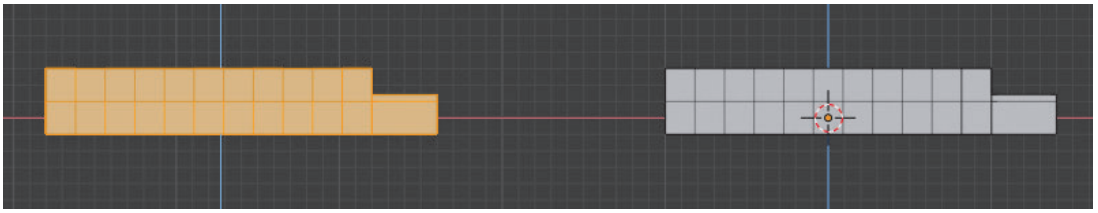
La seconde plateforme n'est plus à la même hauteur que la première (sauf chance !). Nous allons remédier à cela. Utilisez le pavé numérique et appuyez sur la touche 1 ou sur la touche 3 pour vous mettre en vue de côté ou vue de face en fonction de comment sont positionnés vos éléments :

Figure 19.14 : Vue de côté



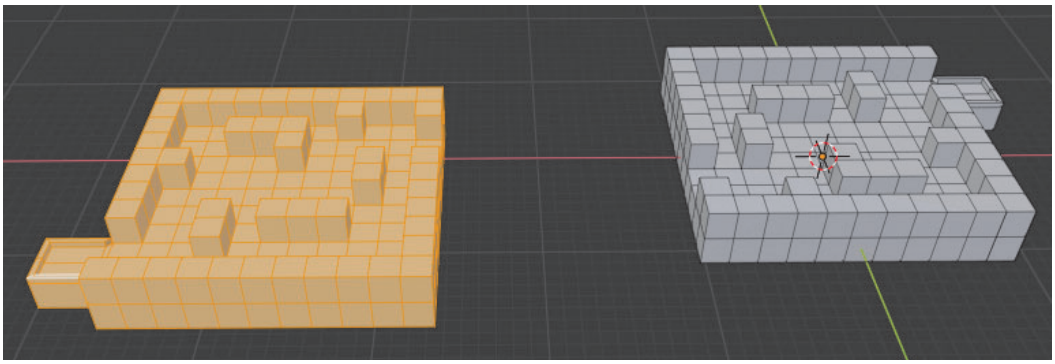
Sur la Figure 19.14, ma caméra est orthographique (ne tient pas compte de la profondeur). Si vous êtes en perspective, utilisez la touche 5 pour passer d'une vue perspective à orthographique et inversement. En vue de côté, il sera plus facile de positionner votre plateforme au même niveau. Appuyez sur G pour déplacer la sélection. Appuyez sur Z pour bloquer le déplacement sur l'axe Z. Déplacez la sélection à la souris et cliquez pour valider.

Figure 19.15 : Mise à niveau



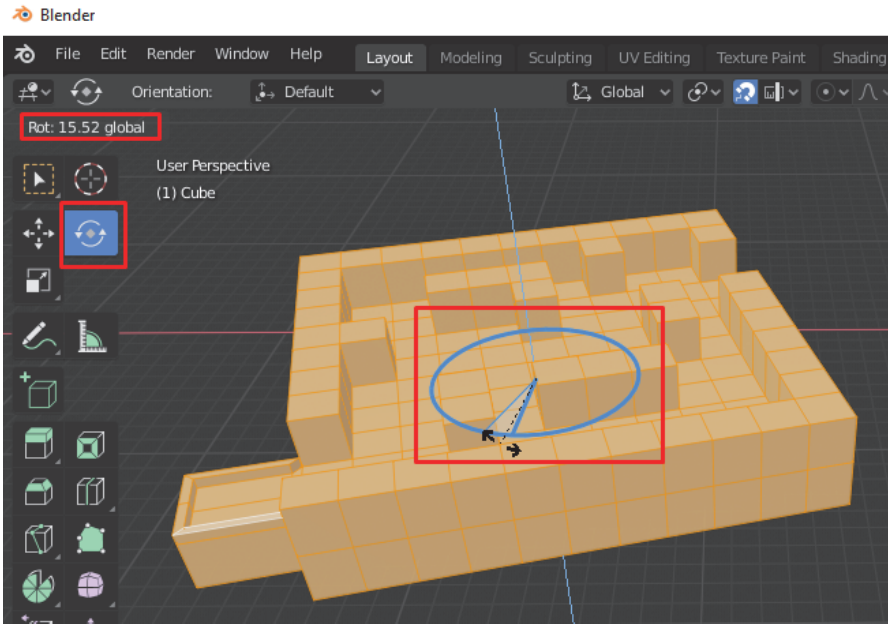
Retournez en vue standard. Pour cela, appuyez sur 5 pour vous mettre en perspective puis appuyez sur la molette de la souris et déplacez la souris pour tourner la vue. Une fois que vous avez une vue qui vous convient, appuyez sur R pour effectuer une rotation puis sur Z pour tourner la forme sur l'axe Z. Tapez 180 pour effectuer une rotation de 180°.

Figure 19.16 : *Rotation de 180°*



Je vous conseille fortement d'utiliser les raccourcis clavier. Si vous avez encore besoin d'éléments visuels, vous pouvez utiliser l'outil de mesure en haut à gauche et l'outil de rotation pour tourner la forme.

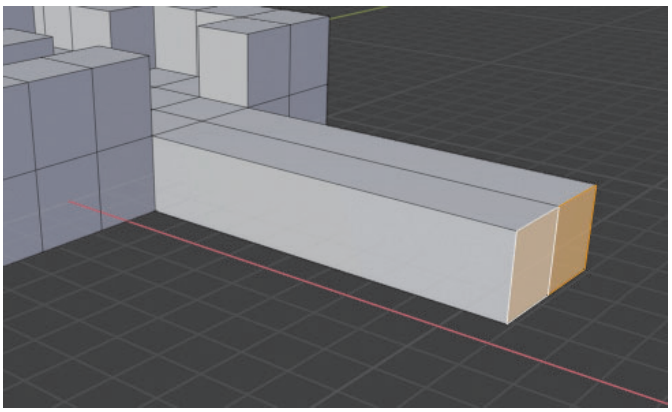
Figure 19.17 : *Rotation visuelle*



19.3. Création des couloirs

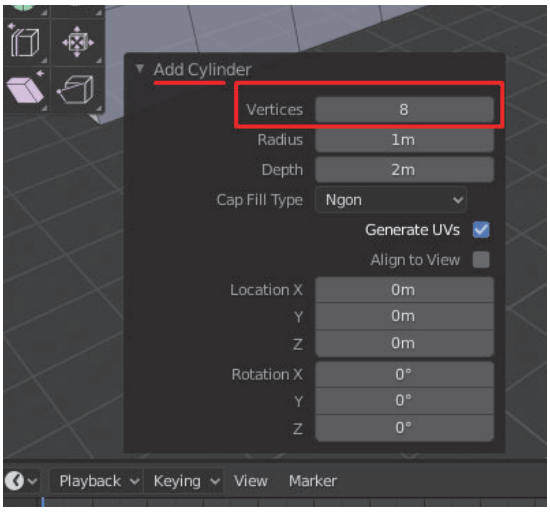
Nous allons maintenant créer le couloir pour relier les deux plateformes. Pour cela, réalisez une extrusion sur l'une des plateformes.

Figure 19.18 : *Création d'un pont*



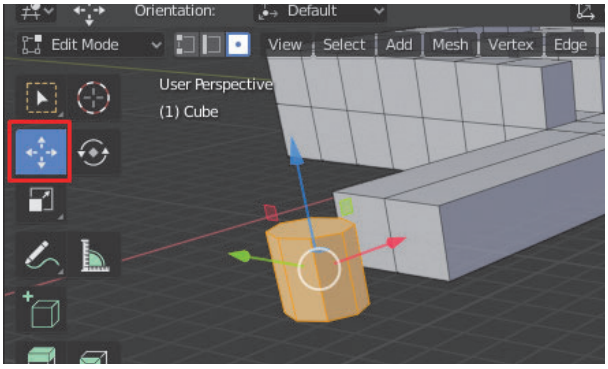
Faites ensuite Maj+A pour ajouter une forme. Choisissez CYLINDER/CYLINDRE pour ajouter un cylindre à la scène. Le cylindre ne sera pas visible car il sera à l'intérieur de la plateforme. Pas de panique. Ouvrez le petit menu déroulant en bas de l'écran pour afficher des paramètres (ces paramètres apparaissent uniquement après création du cylindre avant toute autre manipulation et disparaissent ensuite). Sur ce petit menu, configurez le nombre de VERTICES à 8.

Figure 19.19 : *Modification du cylindre*



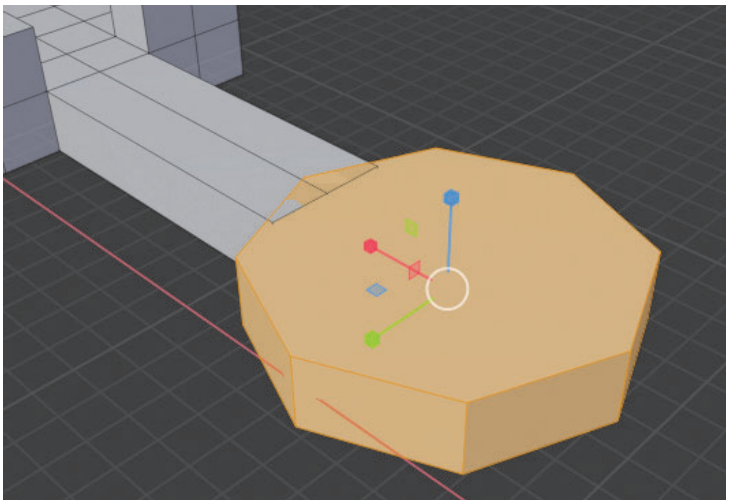
Placez ensuite le cylindre devant le couloir. Vous pouvez utiliser le raccourci G ou encore l'outil de déplacement.

Figure 19.20 : *Déplacement du cylindre*



Agrandissez ce cylindre en X et Y afin de créer une sorte de plateforme intermédiaire. Notez que le cylindre touche le pont car nous souhaitons les relier.

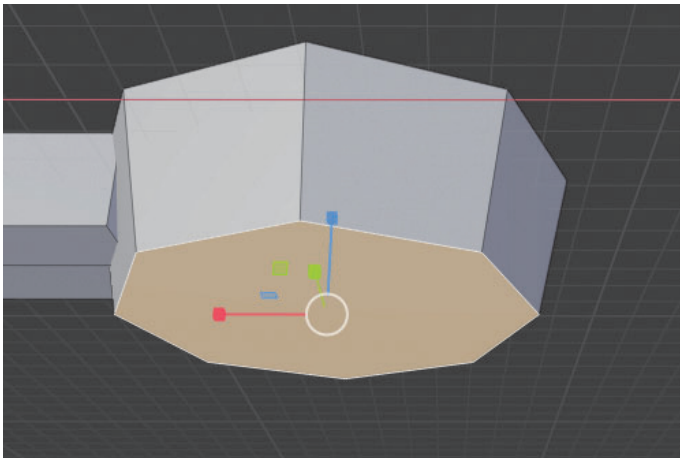
Figure 19.21 : *Agrandir le cylindre*



Vous n'avez pas besoin de faire un copier/coller au millimètre près de ma modélisation. Votre modélisation peut différer, l'essentiel est d'avoir un niveau cohérent praticable. Quoi qu'il arrive, vous disposez des sources avec ce livre, vous pourrez donc ouvrir ma modélisation sous Blender si besoin.

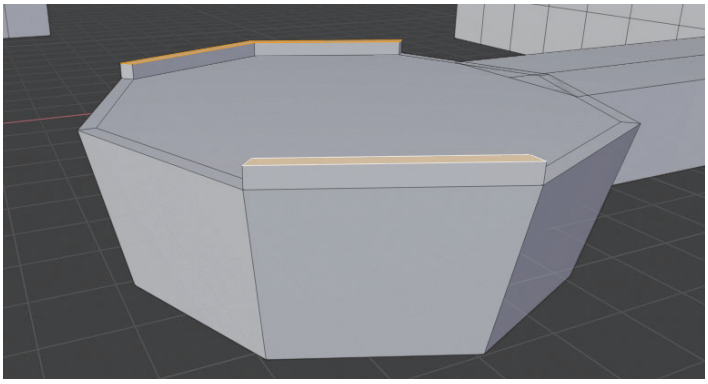
À ce stade, nous allons modifier légèrement le cylindre. Sélectionnez la face du dessous, diminuez la taille de cette face (S) et tirez-la vers le bas (G+Z) pour obtenir quelque chose comme sur la Figure 19.22.

Figure 19.22 : *Modification du cylindre*



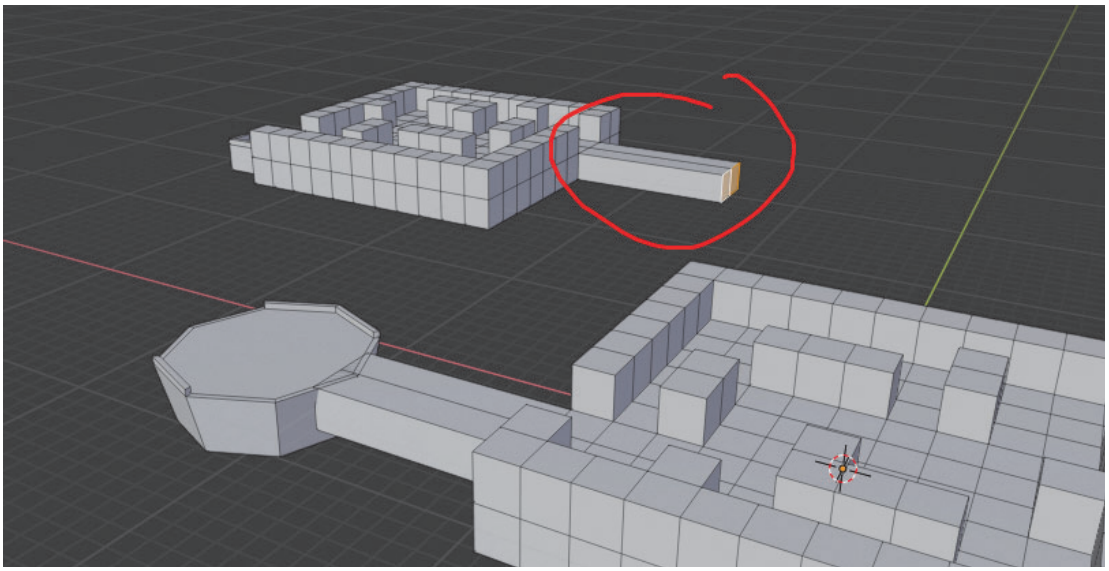
Sélectionnez la face du dessus, appuyez sur la touche I pour incruster des faces et réalisez des extrusions pour créer des petits rebords aléatoires.

Figure 19.23 : *Création de rebords*



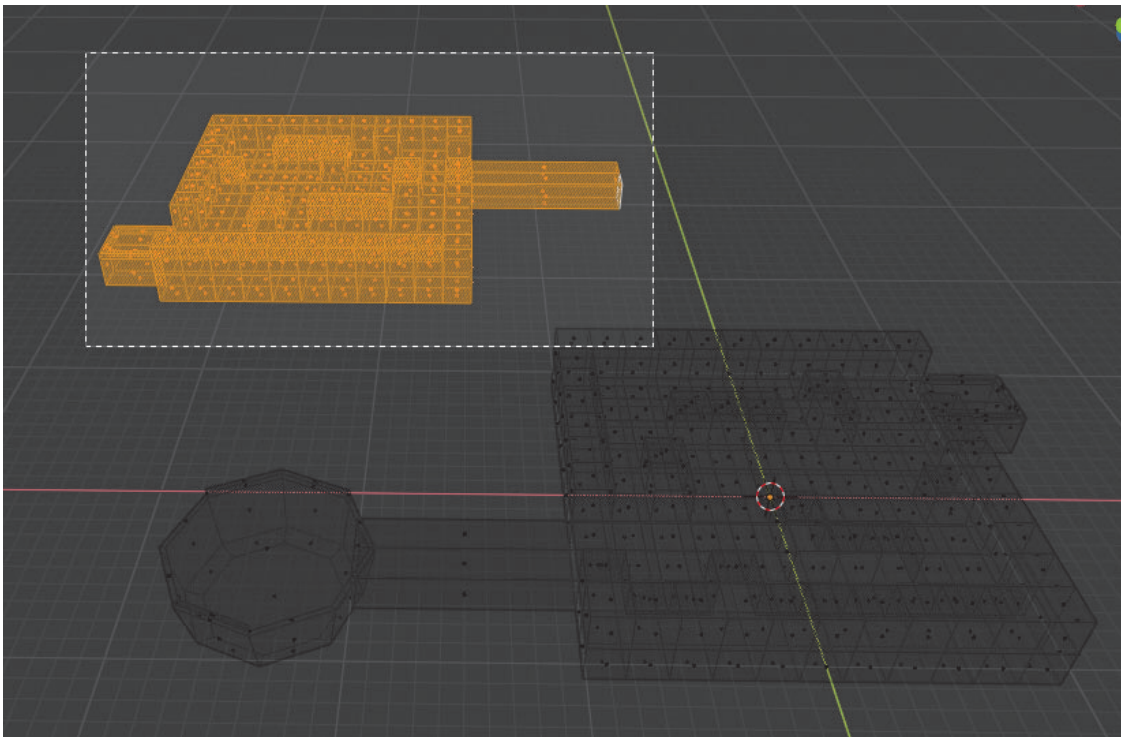
Allez sur l'autre plateforme et créez un couloir comme vous l'avez fait pour l'autre plateforme.

Figure 19.24 : *Création de l'autre couloir*



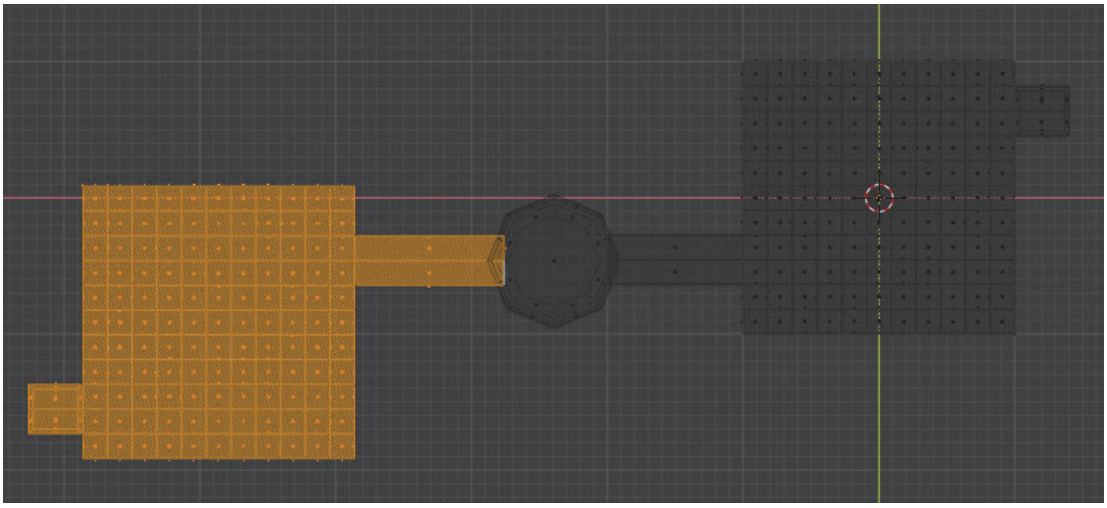
Maintenant passez en transparence (Z) afin de pouvoir effectuer une sélection complète de la plateforme. Utilisez B pour créer une boîte de sélection tout autour de la plateforme.

Figure 19.25 : *Boîte de sélection*



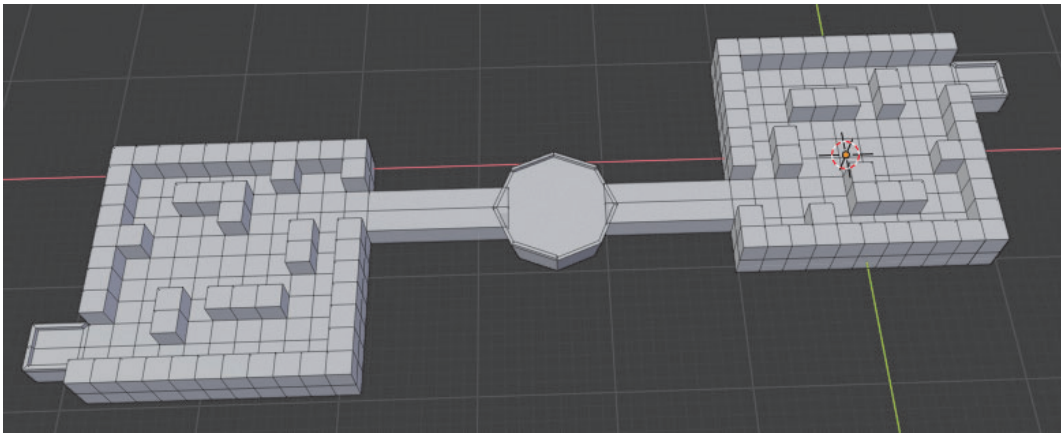
Cela nous permet de sélectionner la plateforme isolée. Utilisez ensuite l'outil de déplacement ou la touche G pour bouger la plateforme afin de la coller au cylindre comme sur la Figure 19.26.

Figure 19.26 : *Relier les plateformes*



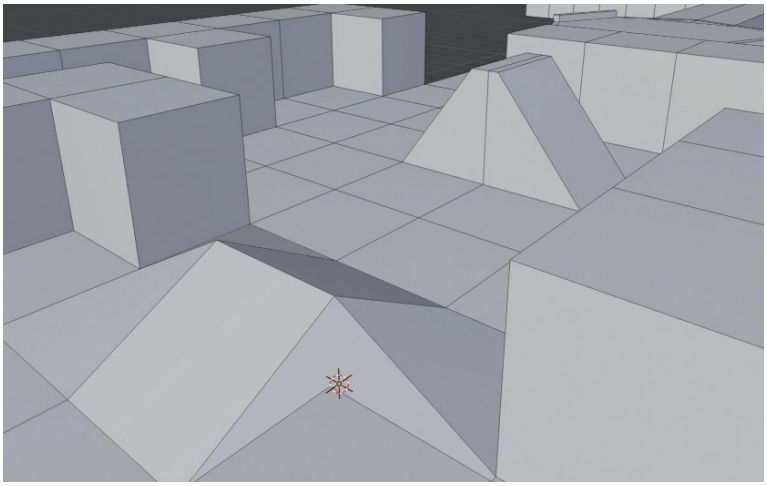
N'hésitez pas à naviguer entre les vues avec par exemple la touche 7 permettant de se mettre en vue de dessus. Adaptez la vue pour être à l'aise lors des manipulations. Une fois terminé, vous aurez la structure globale de votre niveau 1.

Figure 19.27 : *Le niveau 1*



Comme nous avons dupliqué les plateformes, elles sont identiques. Pour apporter quelques variations, modifiez légèrement celle de droite.

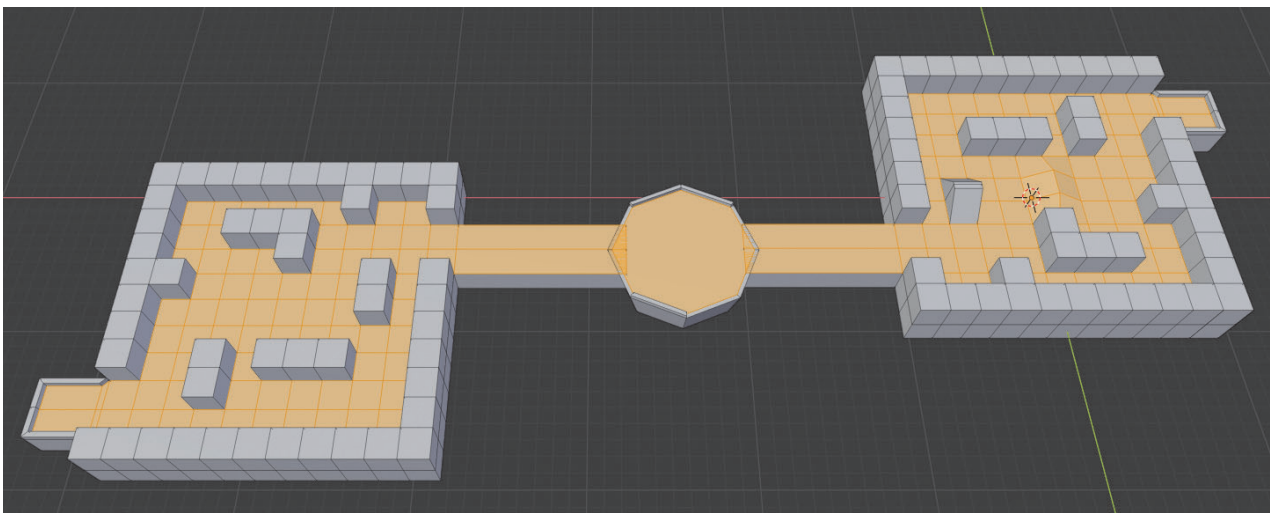
Figure 19.28 : Variations dans le niveau



19.4. Isoler des éléments

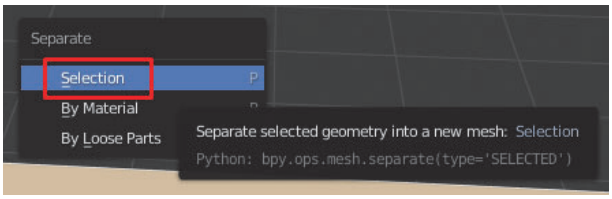
Notre premier niveau est prêt. Nous allons encore isoler le sol praticable des murs pour pouvoir leur appliquer une couleur différente. Pour isoler le sol, sélectionnez toutes les parties sur lesquelles la balle pourra rouler.

Figure 19.29 : Sélection du sol



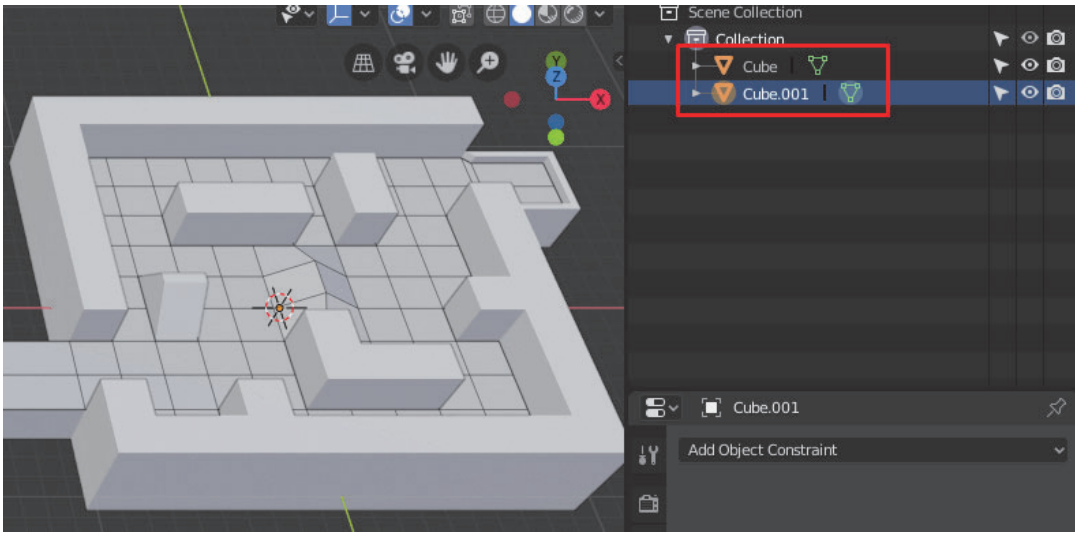
Pour effectuer cette sélection rapidement, utilisez l'outil Cercle de sélection associé à la touche C. Sur la fin, vous pouvez utiliser Maj+Clic pour ajouter à la sélection les quelques faces manquantes. Lorsque vous avez terminé, appuyez sur la touche P. Un menu s'ouvre. Cliquez sur SELECTION pour grouper les éléments sélectionnés.

Figure 19.30 : *Séparer la sélection*



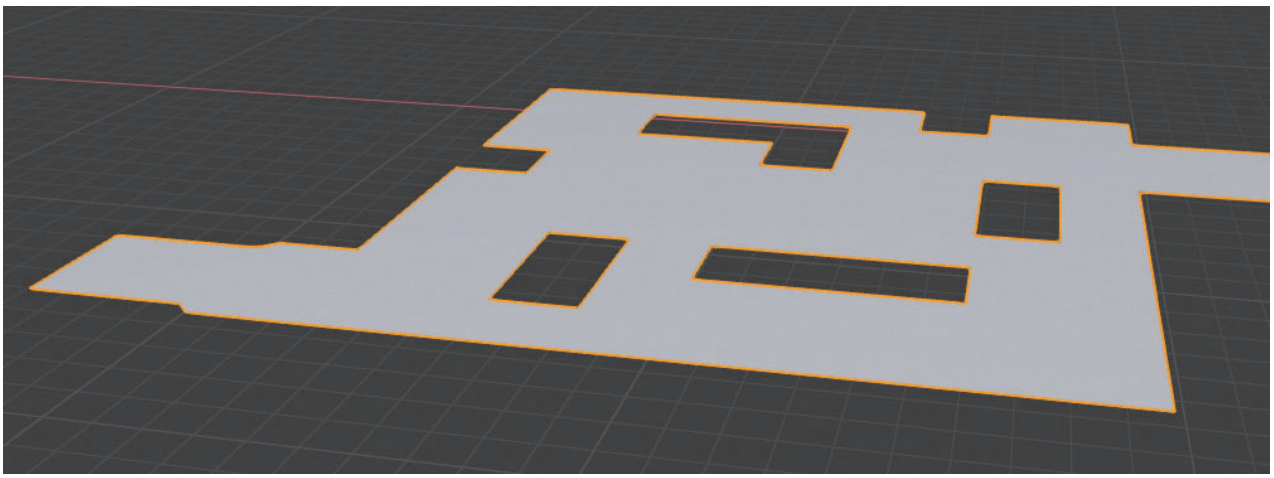
La zone sélectionnée est devenue un objet à part entière, distincte du reste du niveau, comme vous pouvez le voir dans la hiérarchie.

Figure 19.31 : *Séparation du sol et des murs*



Cela vous laissera de la liberté pour faire des manipulations uniquement sur le modèle 3D du sol.

Figure 19.32 : *Le sol séparé du niveau*



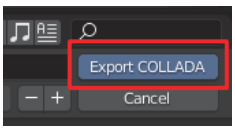
19.5. Exporter pour Godot

Exportez votre niveau 1. Pour cela, cliquez sur FILE/EXPORT et choisissez le format COLLADA (*.dae*). Ce format est pleinement pris en compte par Godot tout comme le format glTF 2 récemment apparu. Le format OBJ peut également faire l'affaire mais pour l'heure, restez sur du Collada qui est plus approprié.

Note › *Depuis la version 4 de Godot, le format glTF 2 est particulièrement mis en avant car il s'agit d'un format ouvert, open-source et pensé pour le transfert.*

Donnez un nom à votre modèle, par exemple `Niveau1`, et cliquez sur le bouton d'exportation (en haut à droite) pour créer le fichier `.dae`.

Figure 19.33 : *Export Collada*



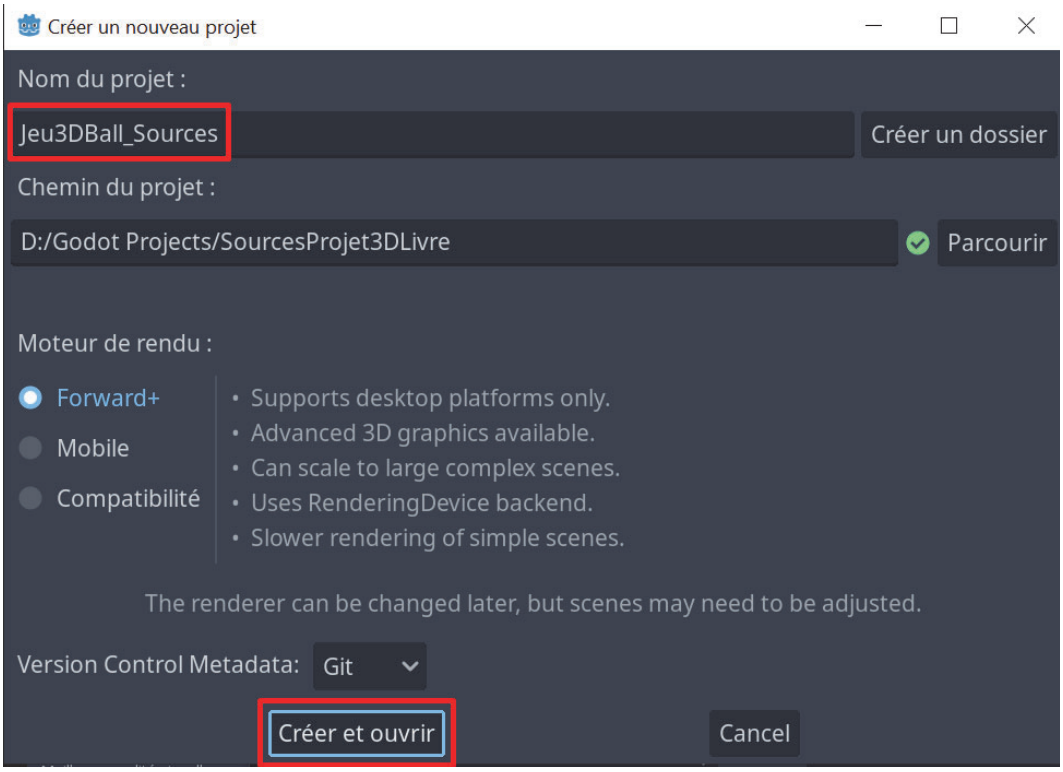
Vous avez réussi à modéliser un niveau 3D et à l'exporter ! Comme vous l'avez vu, avec de simples extrusions, il est possible de modéliser des formes assez complexes. Grâce à cette technique, vous pourrez modéliser des niveaux encore plus complexes. À vous de vous entraîner en réalisant une dizaine de niveaux sous Blender !

20

Mise en place du projet Godot

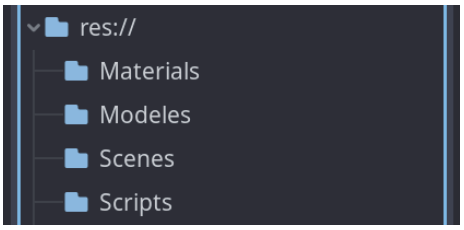
Maintenant que nous avons la modélisation du premier niveau, nous allons l'intégrer au projet Godot. Commencez par créer un nouveau projet. Donnez-lui un nom, placez-vous dans un dossier vide puis cliquez sur CRÉER ET OUVRIR pour l'éditer.

Figure 20.1 : *Création du projet Godot*



Vous vous retrouvez sur une scène vide. Dans le système de fichiers, créez quatre dossiers afin d'organiser votre projet.

Figure 20.2 : *Création des dossiers*

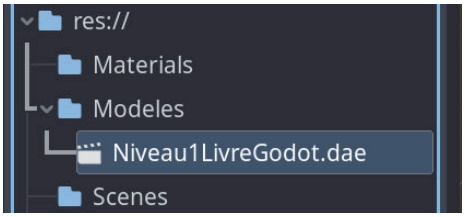


Comme notre jeu sera basique, nous n'aurons pas besoin de plus de dossiers.

20.1. Importation de la modélisation

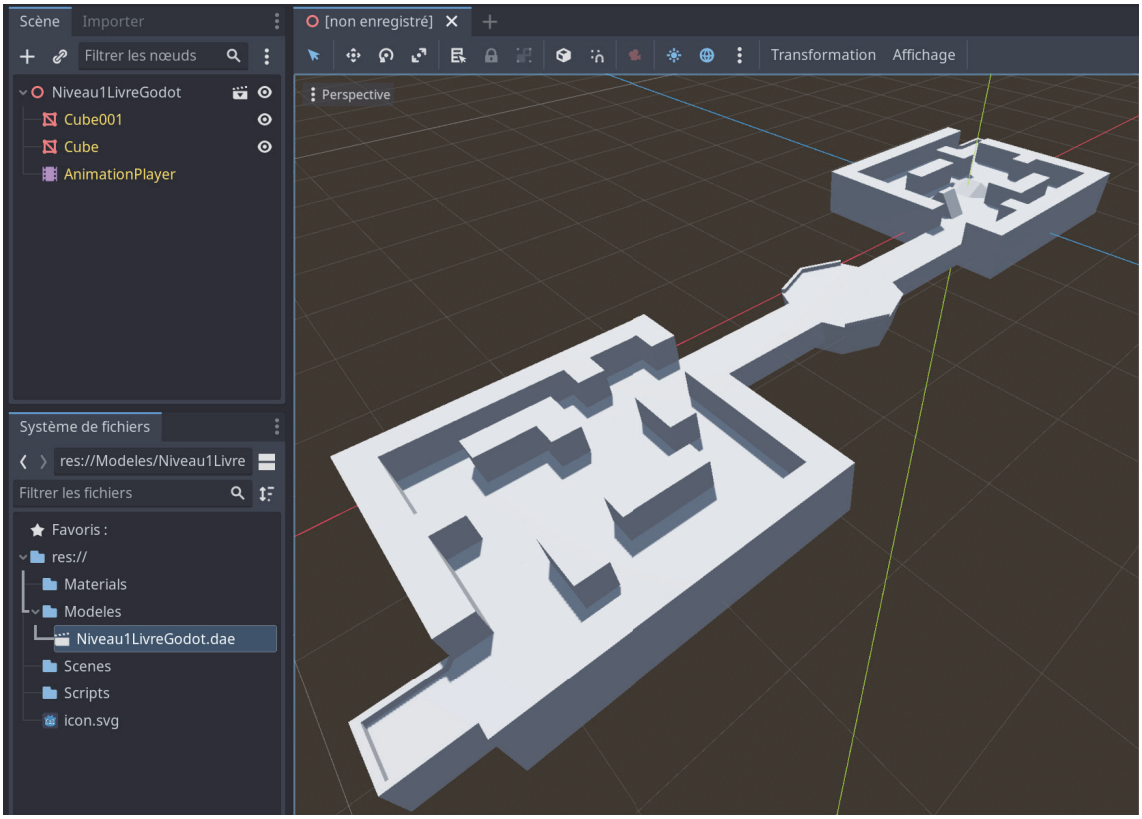
Vous avez maintenant un dossier `Modeles` qui contiendra les modèles 3D de votre jeu. Glissez/déposez dans ce dossier la modélisation 3D du niveau 1 réalisée sous Blender pour l'ajouter au projet Godot. Mon modèle 3D porte le nom de `Niveau1LivreGodot.dae`, vous le retrouverez parmi les sources livrées avec ce livre mais privilégiez votre modélisation plutôt que la mienne.

Figure 20.3 : *Import du modèle*



Faites un clic droit sur votre modèle 3D importé et sélectionnez NOUVELLE SCÈNE HÉRITÉE. Cela aura pour effet de créer une scène à partir de celui-ci.

Figure 20.4 : *Notre modélisation sous Godot*

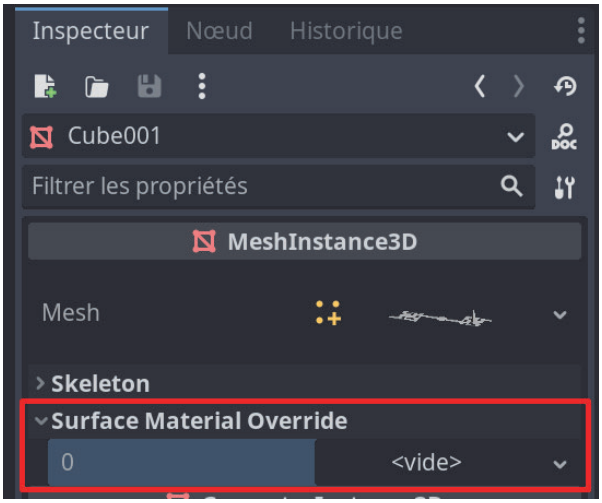


20.2. Coloration du niveau

Nous allons maintenant ajouter un peu de couleur à notre modélisation. La création de couleur (MATERIAL) peut se faire avec un logiciel tiers comme Blender ou directement sous Godot. Mon conseil est de passer par Blender et Photoshop lorsque vous créez des textures complexes et de passer par Godot lorsqu'il s'agit d'une couleur unique.

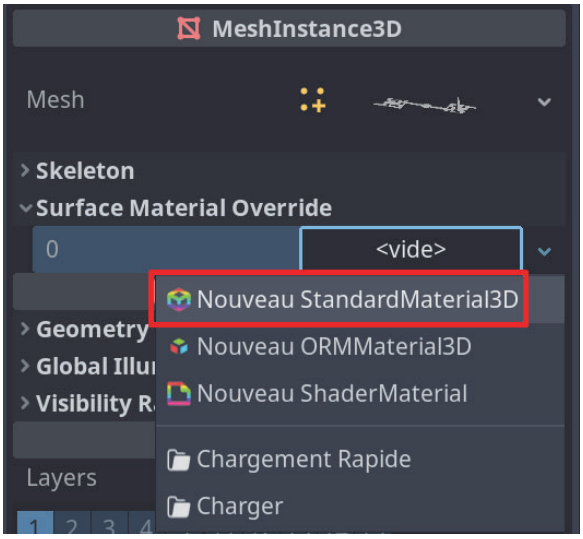
Cliquez sur le sol. Des informations apparaîtront dans l'inspecteur. Ici, dépliez le sous-menu SURFACE MATERIAL OVERRIDE qui permet d'accéder aux propriétés du material et de sa couleur.

Figure 20.5 : *Accès au Material*



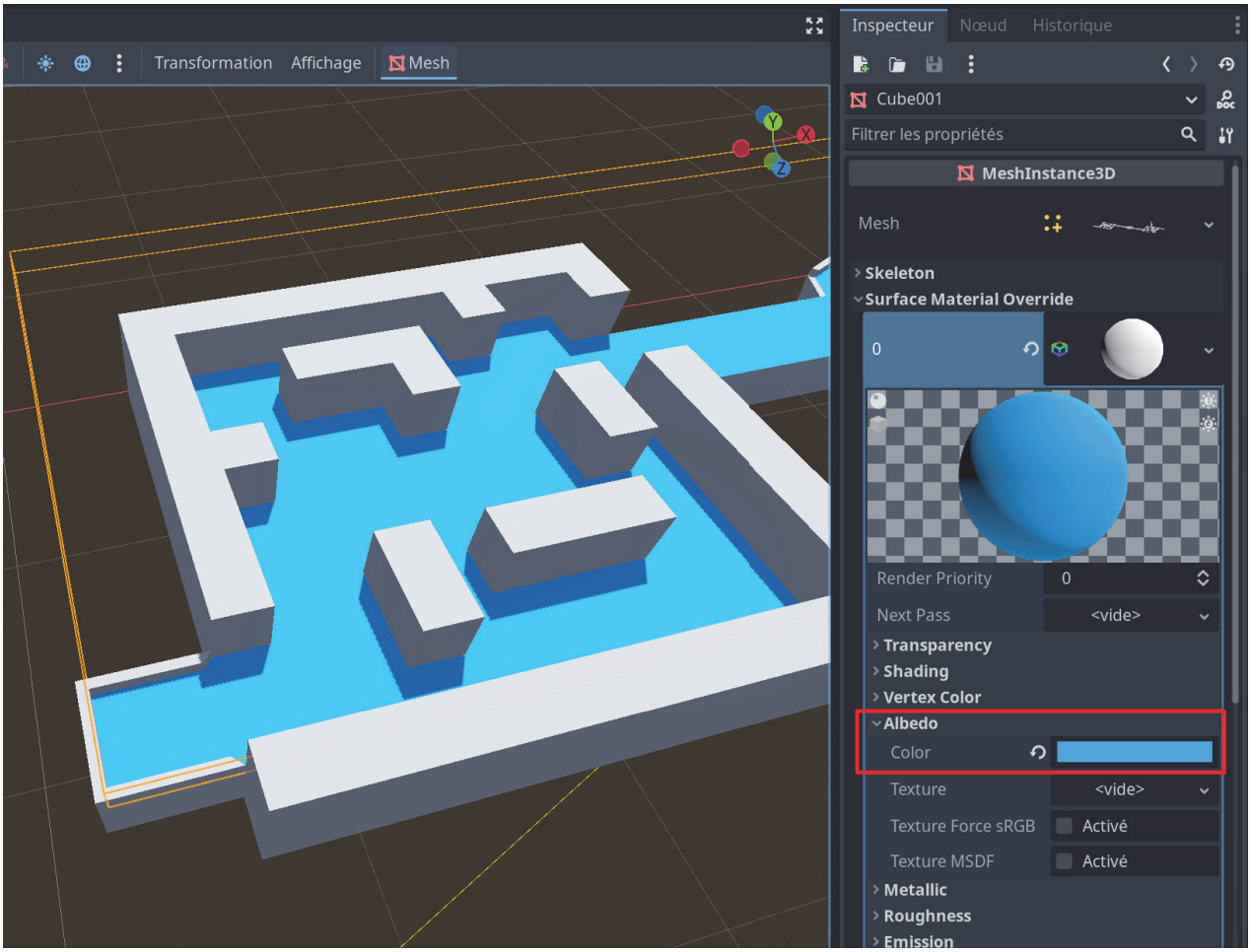
Cliquez ensuite sur VIDE afin de pouvoir créer un NOUVEAU STANDARDMATERIAL3D.

Figure 20.6 : *Création d'un StandardMaterial3D*



Lorsqu'il est créé, ce Material affiche une sphère blanche qui représente la couleur actuelle de votre objet. Cliquez dessus pour afficher les propriétés du Material. Parmi ces nombreuses propriétés, vous trouverez l'ALBEDO qui correspond à la couleur de votre objet. Modifiez la couleur pour colorer le sol comme bon vous semble.

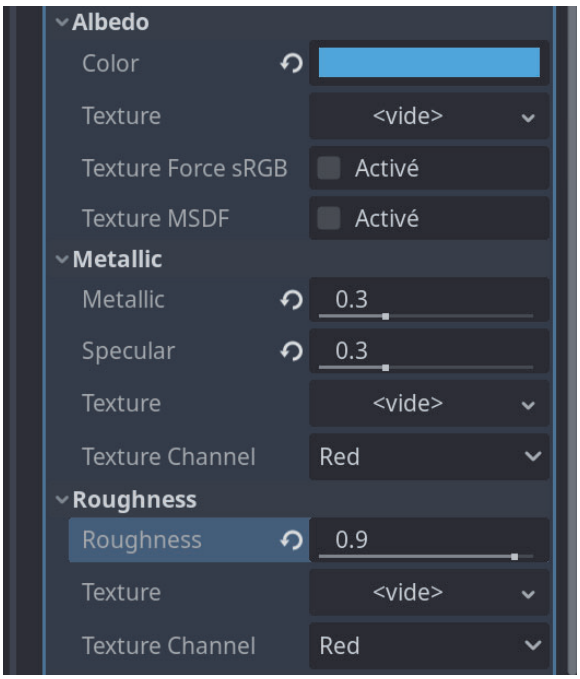
Figure 20.7 : *Colorisation du sol*



Pour ma part, le sol sera teinté en bleu clair. N'hésitez pas à jouer avec les autres paramètres offerts par ce Material. Vous pouvez donner à votre objet un aspect métallique (METALLIC), brillant (SPECULAR), lisse ou rugueux (ROUGHNESS), etc.

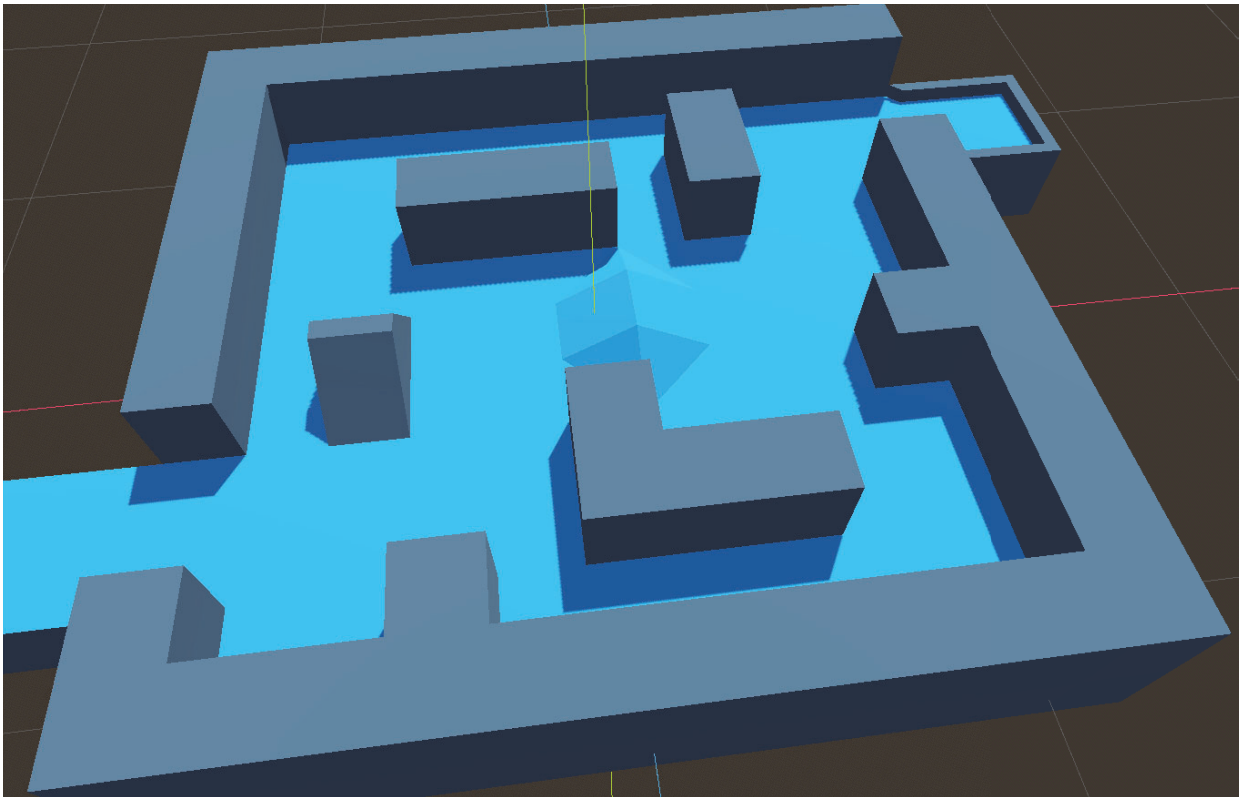
Note › *Je ne rentrerai pas dans les détails de toutes les propriétés du material, il y en a beaucoup trop. Celles que j'ai citées sont les plus utilisées. Retenez simplement que le rôle d'un material est de définir comment doit être affiché un objet et comment l'environnement (par exemple la lumière) doit agir à son contact. En d'autres mots, vous pouvez manipuler la couleur, la transparence, la réflexion de la lumière, la diffraction et plein d'autres propriétés. Si vous voulez plus de détails, je vous invite à consulter la documentation de Godot, par exemple la page Standard Material 3D and ORM Material 3D présente certaines propriétés. Sachez aussi qu'il existe plusieurs types de materials qui ont des propriétés différentes.*

Figure 20.8 : *Quelques paramètres supplémentaires*



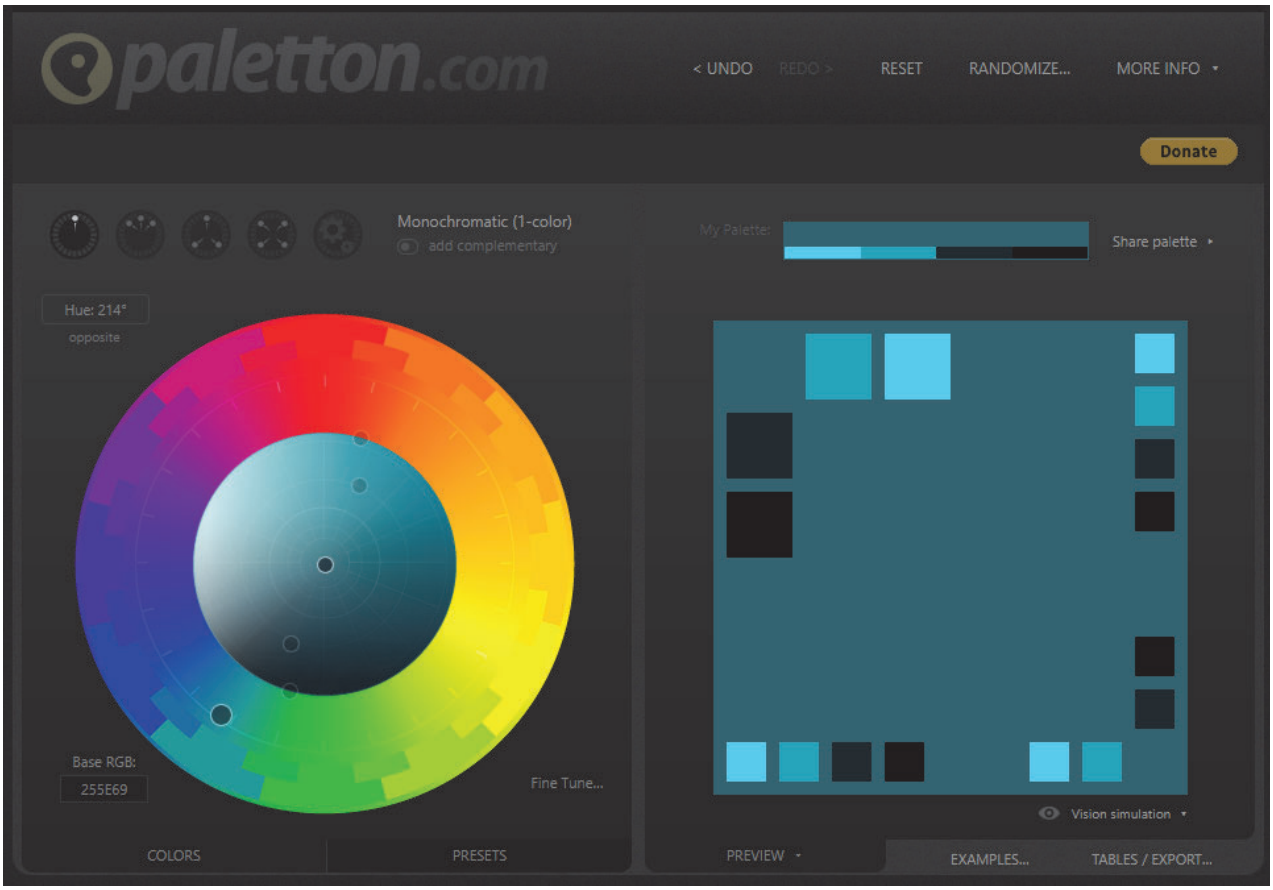
Répétez ces étapes pour colorer les murs de la plateforme.

Figure 20.9 : *Couleur des murs*



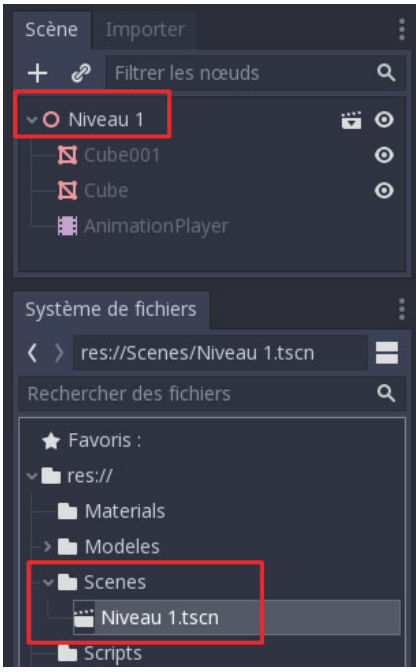
Si vous avez découpé votre modèle en plus de deux parties sous Blender, colorez chaque partie comme bon vous semble. Si vous avez besoin d'aide pour choisir des couleurs qui se combinent correctement, aidez-vous des palettes de couleurs proposées par des sites web comme Paletton.

Figure 20.10 : *Trouver des couleurs qui se mélangent bien*



Pour ne pas perdre votre travail, pensez à sauvegarder régulièrement. D'ailleurs, lorsque vous ouvrez votre modélisation 3D sous forme de scène pour la première fois, il faudra la sauvegarder dans le dossier des scènes pour bien conserver vos modifications. Renommez votre nœud spatial en `Niveau 1` puis enregistrez la scène dans le dossier des scènes comme cela :

Figure 20.11 : *Sauvegarde du niveau*



20.3. Ajout de la physique

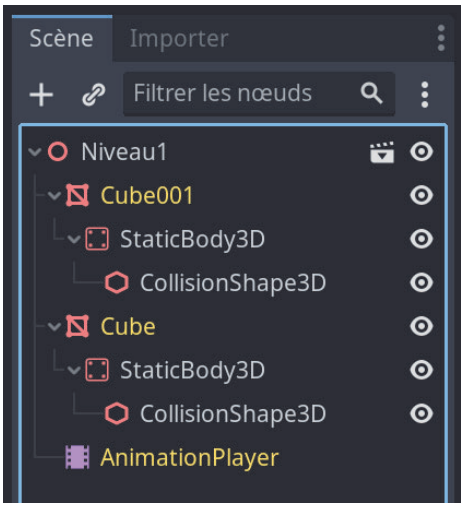
Nous avons besoin de physique pour notre jeu. La balle doit rouler sur le niveau, il faut donc que ce niveau soit solide et que les collisions entre la balle, le sol et les murs soient prises en compte.

Notre niveau sera statique, il ne bougera pas. Nous allons donc créer un StaticBody et un CollisionShape. Vous pouvez faire cela de façon classique en ajoutant les nœuds manuellement comme nous l'avons fait pour le jeu 2D ou utiliser un générateur de collisions, comme nous allons le voir maintenant. Cliquez sur le sol de votre niveau puis sur le bouton MESH [maillage] en haut de l'écran pour sélectionner CRÉER UN CORPS STATIQUE TRIMESH. Cette manipulation a pour effet de créer un StaticBody ainsi qu'un CollisionShape pour votre objet. Le composant Collision épousera parfaitement la forme de votre modèle 3D.

Figure 20.12 : *Création du maillage*

Répétez cette étape pour la seconde partie du modèle 3D. Consultez ensuite l'arbre de votre scène pour vérifier la présence des composants ajoutés.

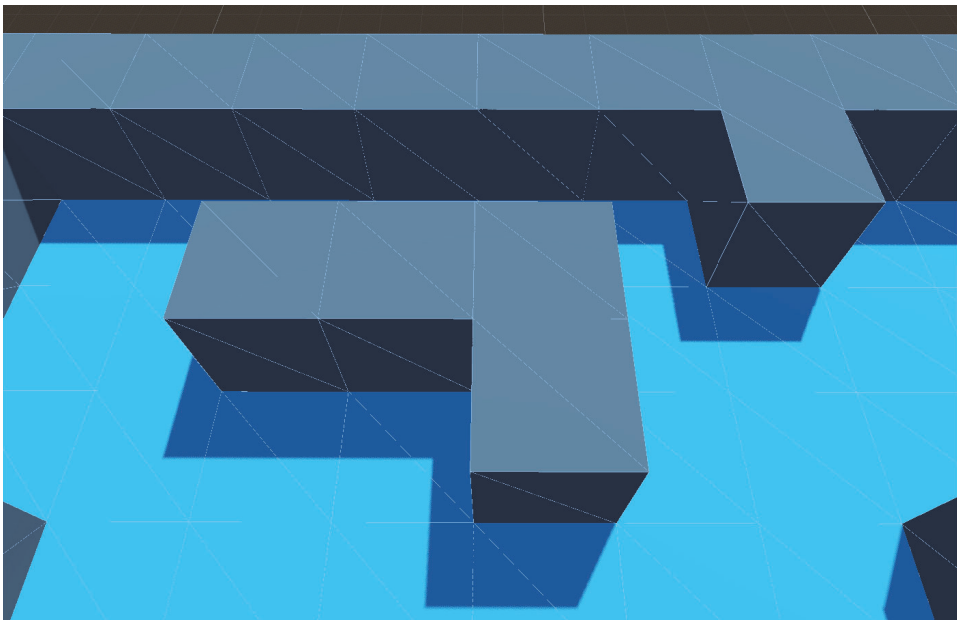
Figure 20.13 : *Les composants de la scène*



Si vous examinez le modèle 3D de plus près, vous verrez que le maillage de la collision est visible (voir Figure 20.14).

Note *› L'utilisation de ce générateur de collisions est très pratique car simple et rapide. Sachez qu'il est aussi possible de créer des collisions directement sous Blender en suffixant le calque correspondant avec* `-col`*. Pour plus d'infos, consultez la documentation associée.*

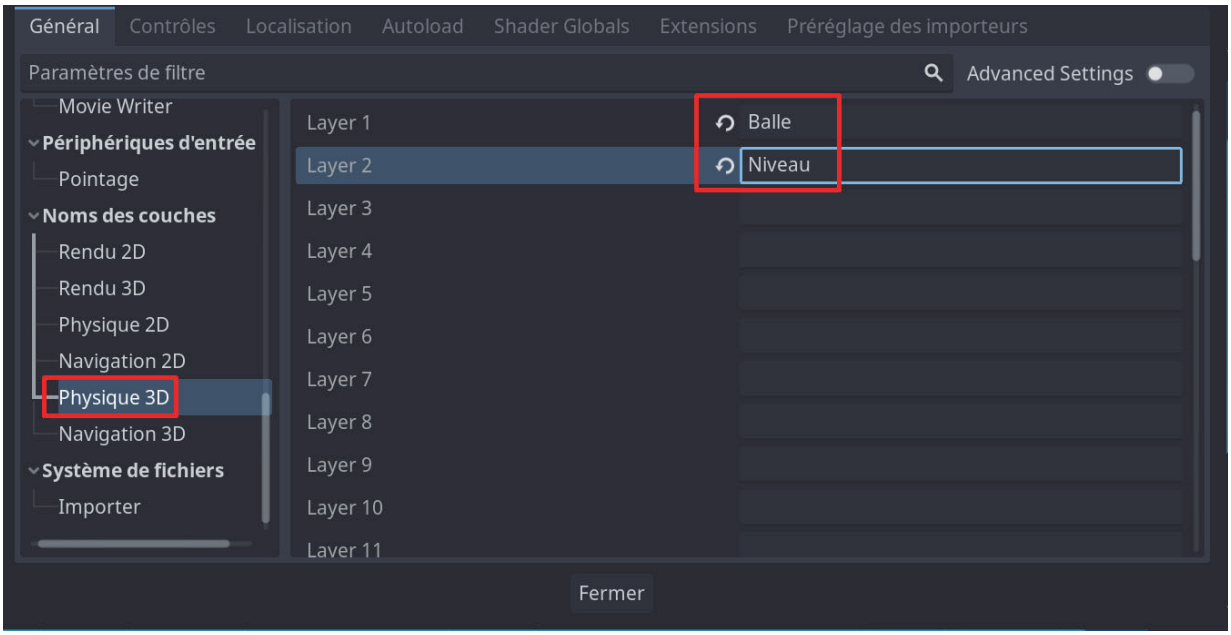
Figure 20.14 : *Maillage de la collision*



Nous avons presque terminé la configuration de notre niveau. Nous devons encore créer des calques et les associer afin que les collisions entre la balle et le décor soient prises en compte. Nous ne détaillerons pas la manipulation, reportez-vous aux chapitres Mise en place d'un TileSet et Interaction avec les objets de la partie Développement d'un jeu 2D pour plus de précisions.

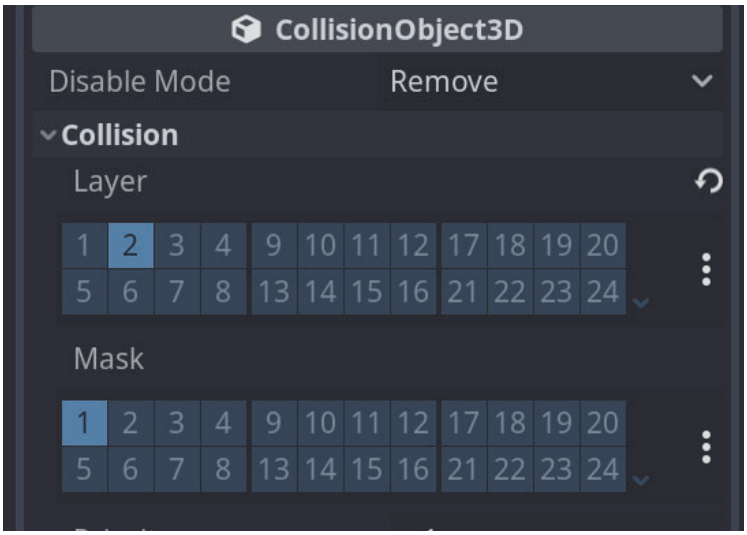
Allez dans les paramètres du projet, sous la rubrique PHYSIQUE 3D, afin d'ajouter les calques `Balle` et `Niveau`.

Figure 20.15 : *Ajout des layers*



Retournez sur votre modèle 3D, cliquez sur le StaticBody du sol, dépliez la section COLLISION et placez le sol sur le calque `Niveau` puis faites-le interagir avec le masque `Balle`.

Figure 20.16 : *Configuration de la collision*



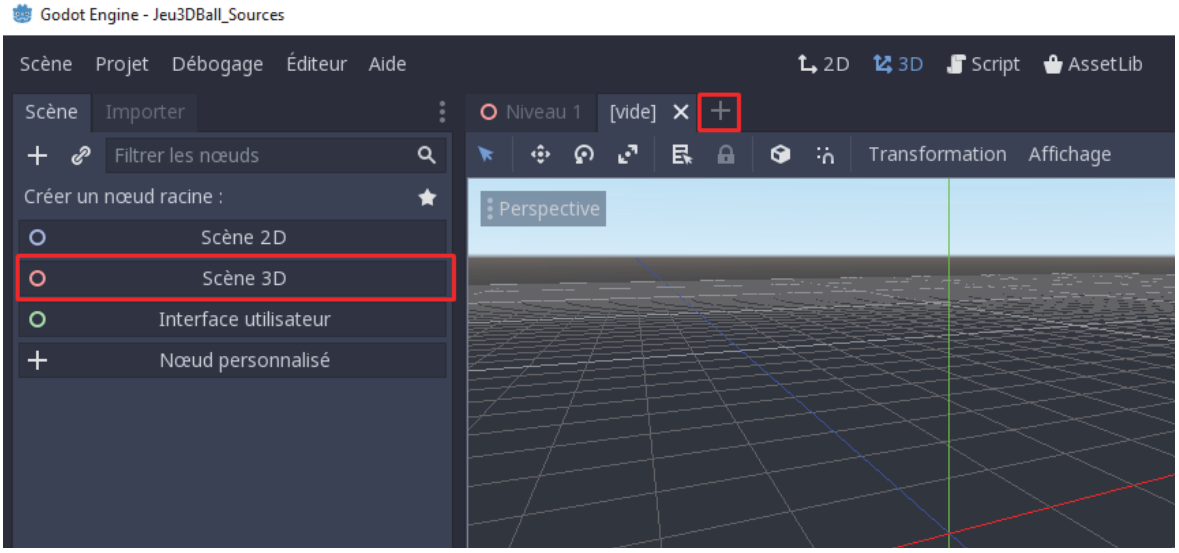
Pensez à faire cette manipulation également pour les murs de votre plateforme (en effet notre modèle 3D est découpé en deux modèles qu'il faut configurer). Puis enregistrez votre scène.

20.4. Création de la scène du premier niveau

Vous venez de configurer le modèle de votre niveau 1. Il est temps de créer la scène du niveau. Cette scène regroupera le niveau, la balle, les objets à ramasser et l'interface.

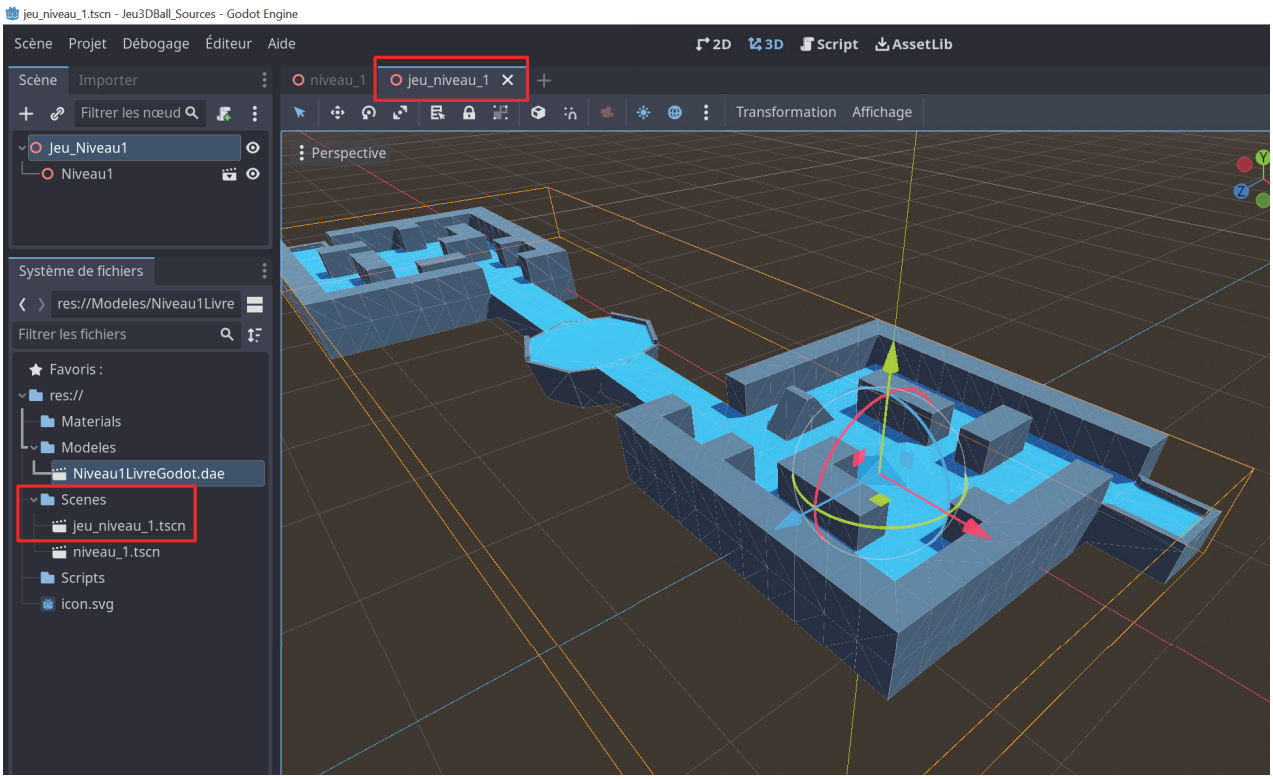
Créez une nouvelle scène et cliquez sur SCÈNE 3D afin de créer un nœud spatial vide qui sera le parent de tous les autres objets.

Figure 20.17 : *Création d'une scène*



Renommez le nœud principal en `Jeu_Niveau1`, instanciez la scène `Niveau1` comme vous savez le faire et enregistrez votre scène.

Figure 20.18 : *Création de la scène principale*



C'est sur cette scène que vous allez créer votre jeu à proprement parler.

Cela clôt ce chapitre. Notre modèle 3D du niveau 1 est configuré et la scène principale du premier niveau créée, prête à accueillir la balle et les éléments à ramasser. Dans le chapitre suivant, nous allons créer la balle.

21

Création de la balle

Pour pouvoir tester que tout fonctionne bien, nous avons maintenant besoin de créer la balle qui sera le personnage principal du jeu. Comme nous la réutiliserons dans tous les niveaux, nous la créerons dans une scène indépendante et nous l'instancierons dans le niveau 1 et tous les autres niveaux du jeu.

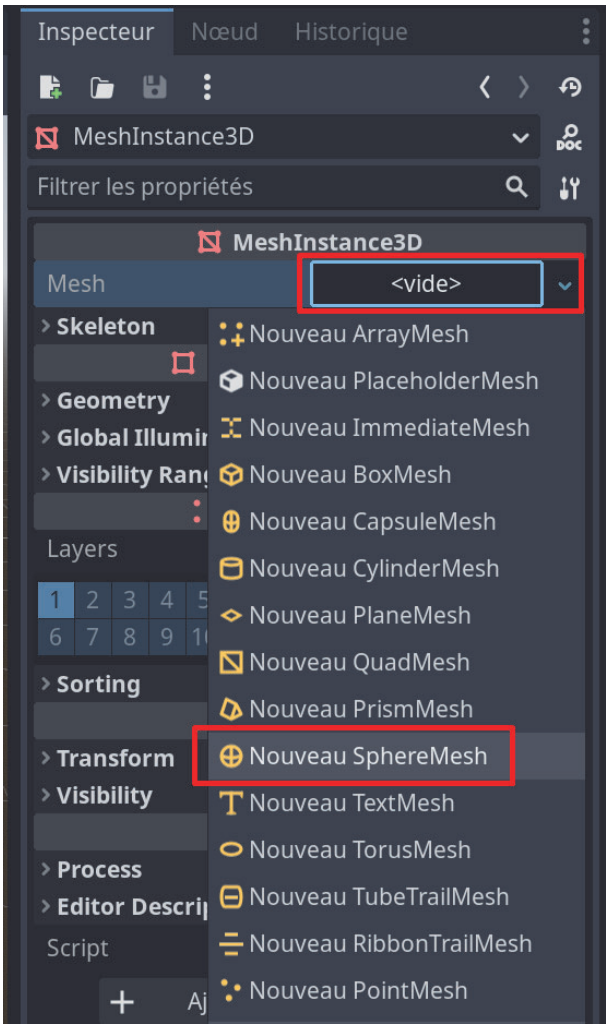
21.1. Création de l'objet

Créez une nouvelle scène. Sur cette scène, ajoutez un RigidBody3D. Il s'agit du principal composant de notre balle. Ce composant permettra de lui ajouter de la gravité et de la faire rouler sur la plateforme. Nous ajoutons ensuite un MeshInstance3D.

Figure 21.1 : *Préparation de la balle*

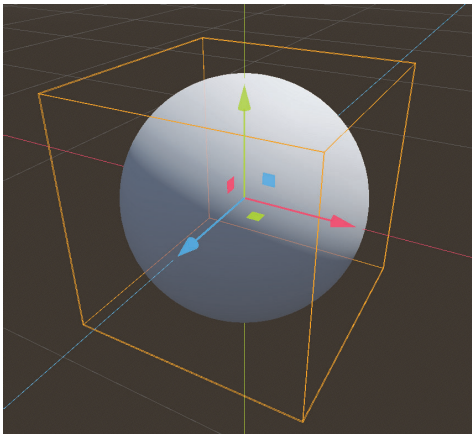
Via l'inspecteur, ajoutez un Mesh de type Sphère à votre composant.

Figure 21.2 : *Ajout du Mesh*



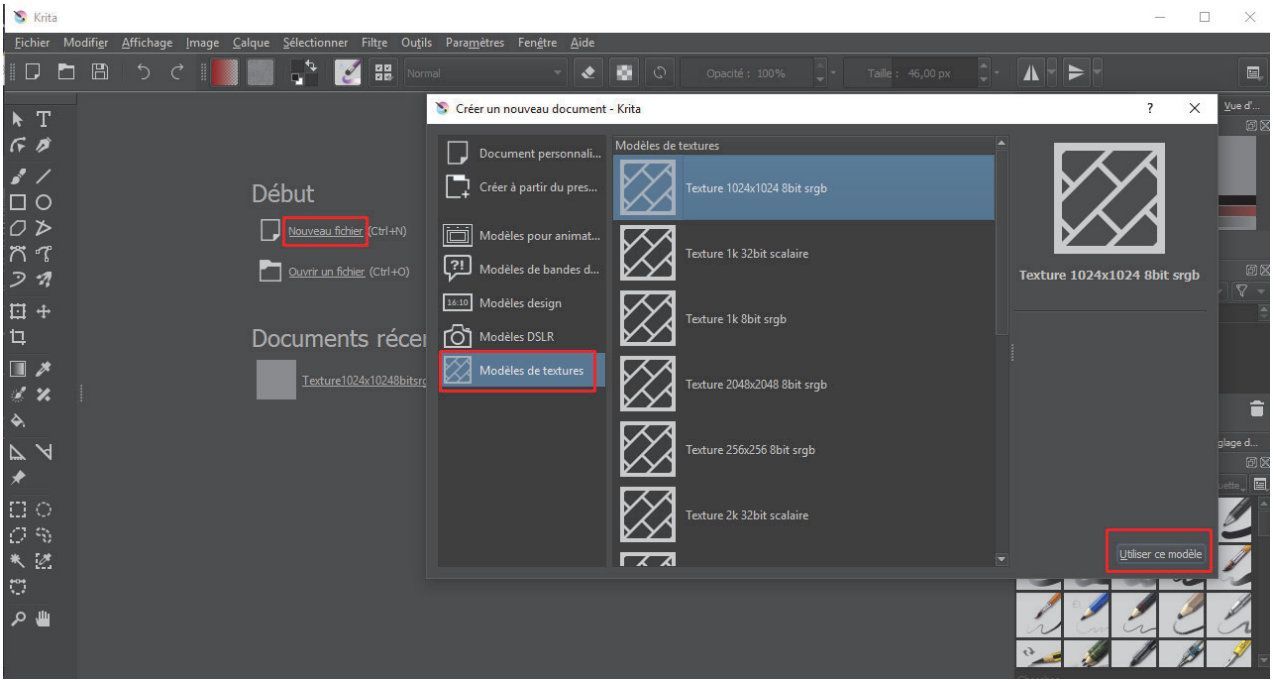
Une fois ceci fait, la balle apparaît.

Figure 21.3 : *Balle de base non texturée*



Comme pour le modèle 3D du niveau, nous allons la colorer. Créez un StandardMaterial3D puis, dans les propriétés de ce Material, modifiez l'ALBEDO pour colorer la balle. Cette fois, je vous propose de créer une texture. En effet, si la balle est de couleur unie, nous n'aurons pas la sensation du roulement, alors qu'avec une texture, nous la verrons tourner. Pour créer une texture, utilisez le logiciel de votre choix. Je vous conseille Krita qui est un logiciel libre. Vous pouvez télécharger la version compatible avec votre système d'exploitation sur le site officiel. Vous pouvez même télécharger une version portable ne nécessitant pas d'installation. Une fois téléchargé, lancez Krita et créez une nouvelle texture.

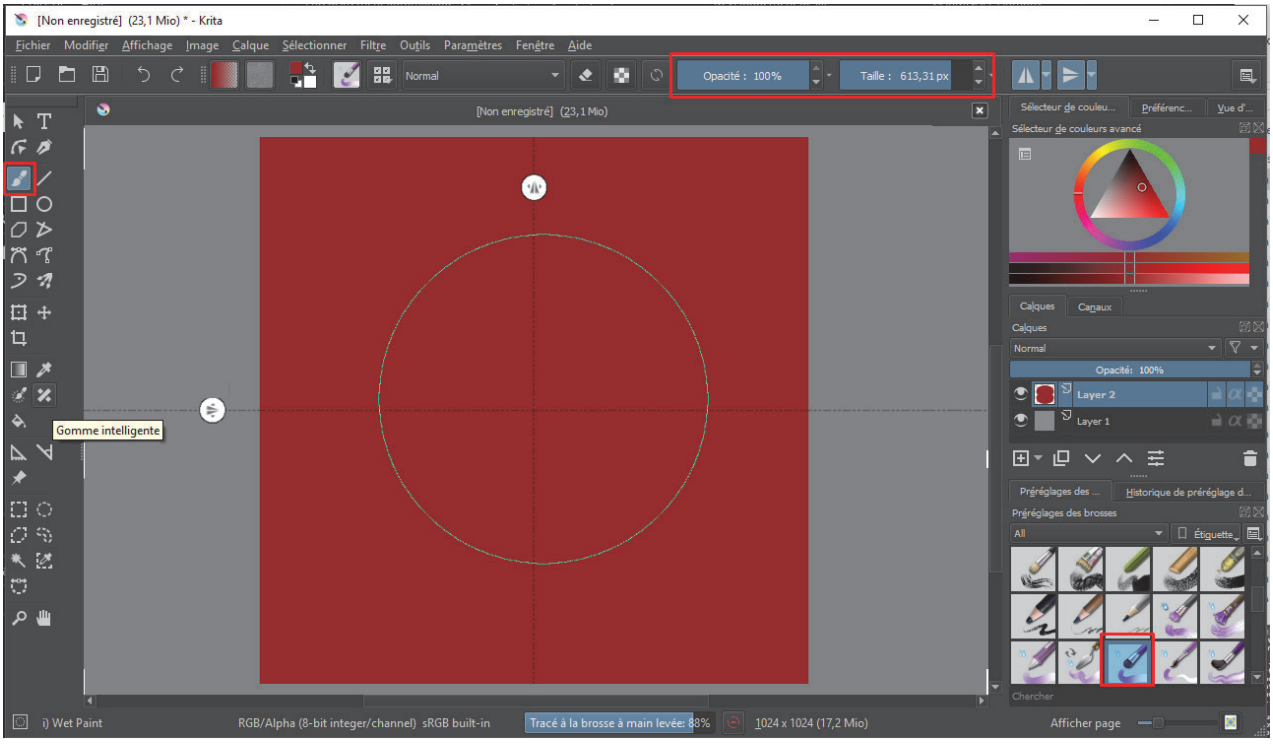
Figure 21.4 : *Création d'une texture*



Lorsque votre texture est créée, activez le mode miroir en vertical ainsi qu'en horizontal . Ce mode miroir permet de dessiner en symétrie. De cette façon, si votre pinceau sort de la texture, le trait qui sort sera répété en horizontal et vertical. Cela nous sera utile car la texture sera appliquée sur un modèle 3D et pourra ainsi se *replier* en 3D : les traits se relieront dans la continuité. C'est une texture répétable.

Commencez par sélectionner un pinceau et une couleur rouge. Agrandissez la taille du pinceau pour peindre entièrement votre texture en rouge.

Figure 21.5 : *Couleur de fond*



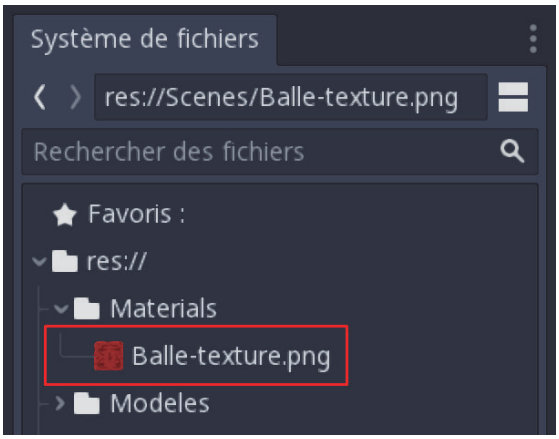
Une fois que vous avez une couleur de fond, changez de couleur par un ton plus foncé et réduisez la taille du pinceau. Vous pouvez ensuite réaliser un petit gribouillage pour avoir un motif.

Figure 21.6 : *Texture élémentaire*

Note *› Les éditions D-BookeR proposent un livre sur l'utilisation de Krita si vous souhaitez aller plus loin avec ce logiciel.*

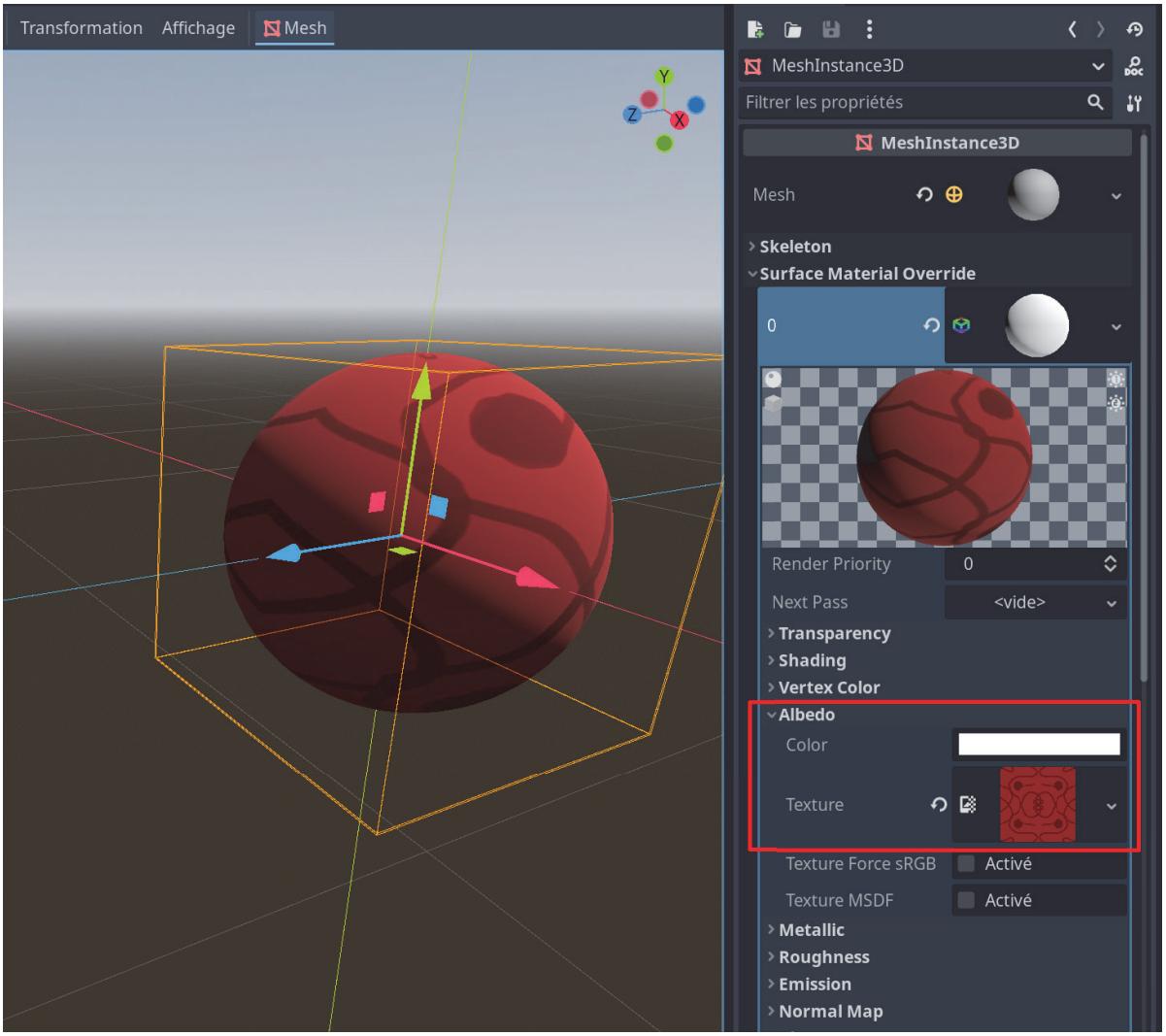
Lorsque vous avez terminé, exportez votre texture en PNG en cliquant sur FICHIER/EXPORTER et sauvegardez le fichier sur votre bureau. Importez ensuite votre PNG dans Godot.

Figure 21.7 : *Import de la texture*



Vous pouvez maintenant appliquer cette texture à votre Material.

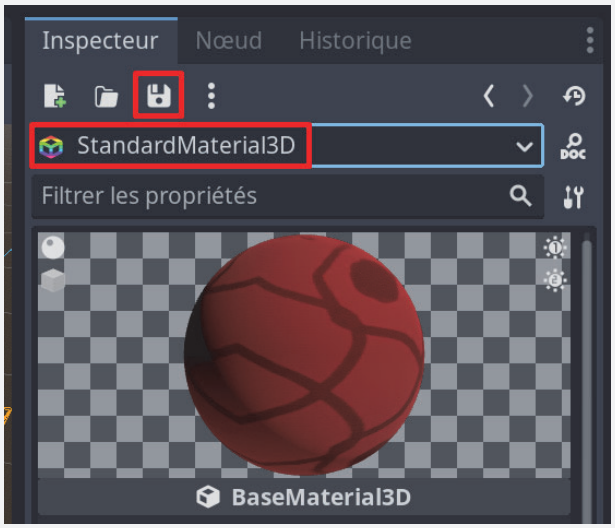
Figure 21.8 : *Texture sur la balle*



RÉUTILISEZ VOS COMPOSANTS MATERIAL

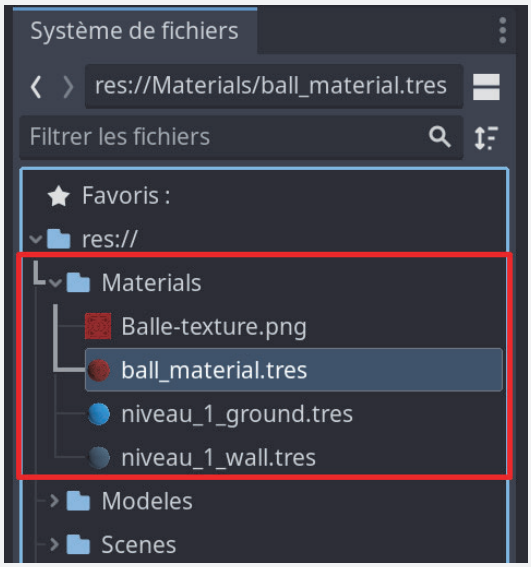
Vous pouvez sauvegarder chacun de vos Material et les conserver dans le système de fichiers. Cela vous permettra de les réutiliser. Pour cela, sélectionnez la ressource que vous voulez enregistrer. Dans notre cas il s'agit du Material. Puis cliquez sur la disquette pour pouvoir enregistrer votre ressource dans le système de fichiers.

Figure 21.9 : *Enregistrement du Material*



Vous pouvez sauvegarder tous les materials du projet. Pour une bonne organisation, je vous invite à les enregistrer dans le dossier `Materials` que nous avons créé.

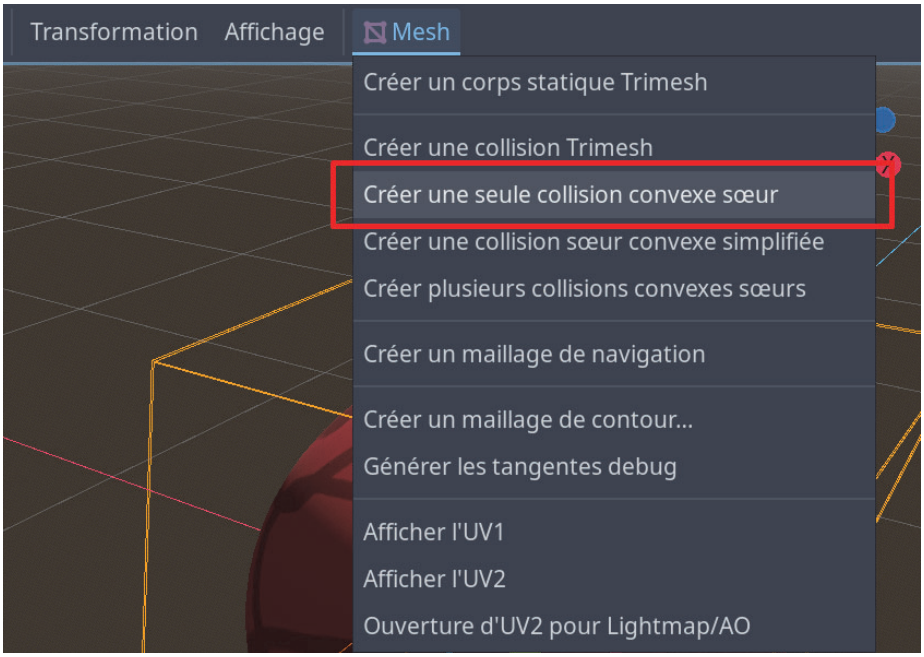
Figure 21.10 : *Dossier Materials*



21.2. Préparation des collisions

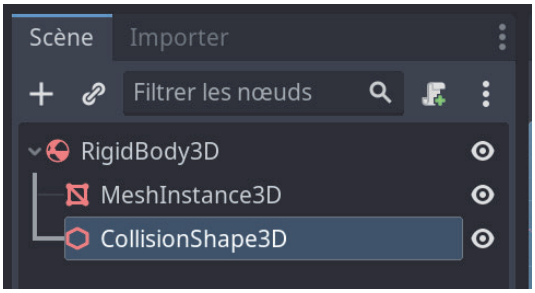
Sélectionnez votre Mesh (MeshInstance3D de la balle) afin de faire apparaître le bouton MAILLAGES qui va permettre de générer votre composant de collision. Cliquez sur ce bouton et choisissez CRÉER UNE COLLISION CONVEXE SOEUR.

Figure 21.11 : *Création de la collision*



Cela a pour effet de créer juste un objet de collision (CollisionShape3D). Il nous sera utile pour gérer les collisions entre la balle et les décors.

Figure 21.12 : *Création de la collision*



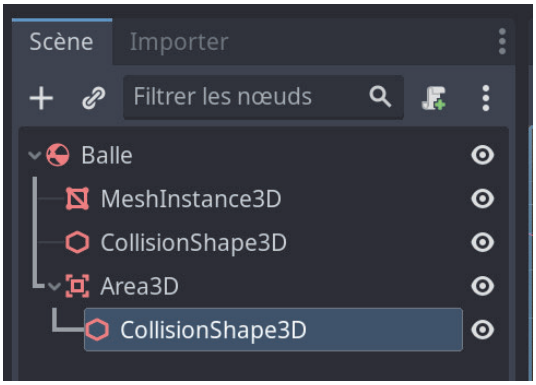
Pour que vos collisions soient prises en compte, pensez à paramétrer le RigidBody via l'inspecteur afin que la balle interagisse avec les calques nécessaires.

Figure 21.13 : *Collisions*

Tant que vous êtes sur le RigidBody, renommez-le en `Balle`, ce sera plus parlant.

Notre balle est désormais fonctionnelle. Nous pourrions programmer ses mouvements, mais nous allons poursuivre sa configuration car nous aurons besoin de deux autres composants pour gérer cette fois les collisions avec les objets à ramasser : un Area et un second CollisionShape, enfant de l'Area, qui renseignera ce dernier sur les zones de la balle pouvant entrer en collision. Créez donc un second CollisionShape identique au premier (donc avec un SphereShape3D). Créez ensuite un Area. Faites glisser le second CollisionShape dans l'Area pour obtenir l'arbre de la Figure 21.14. Nous avons donc d'une part un CollisionShape qui rend la balle solide, d'autre part un Area et un CollisionShape qui la rendent capable de ramasser des objets.

Figure 21.14 : *Arbre final de la balle*

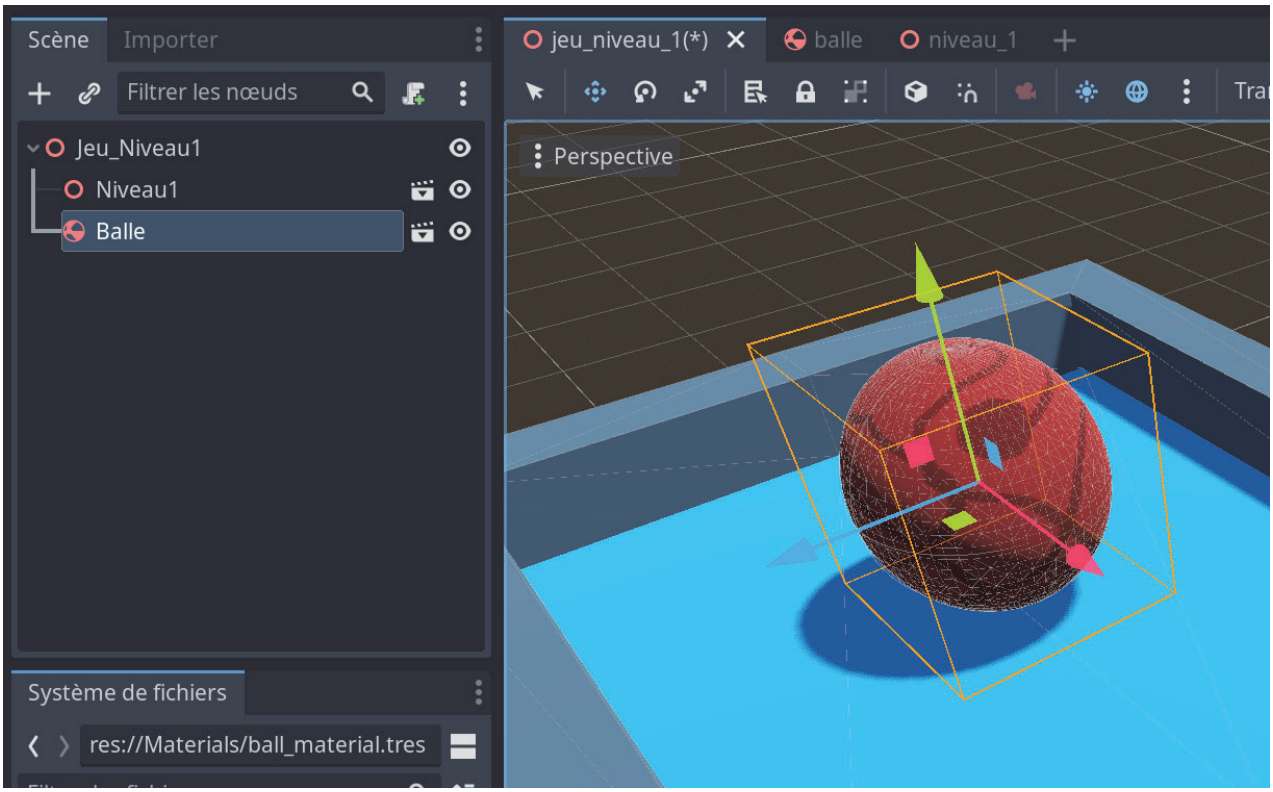


Nous reviendrons plus tard sur la scène de la balle, lorsque nous aurons créé des calques pour les objets à ramasser, pour spécifier les collisions de l'Area. Sauvegardez la scène et nommez-la `Balle`.

21.3. Instanciation de la balle et redimensionnement

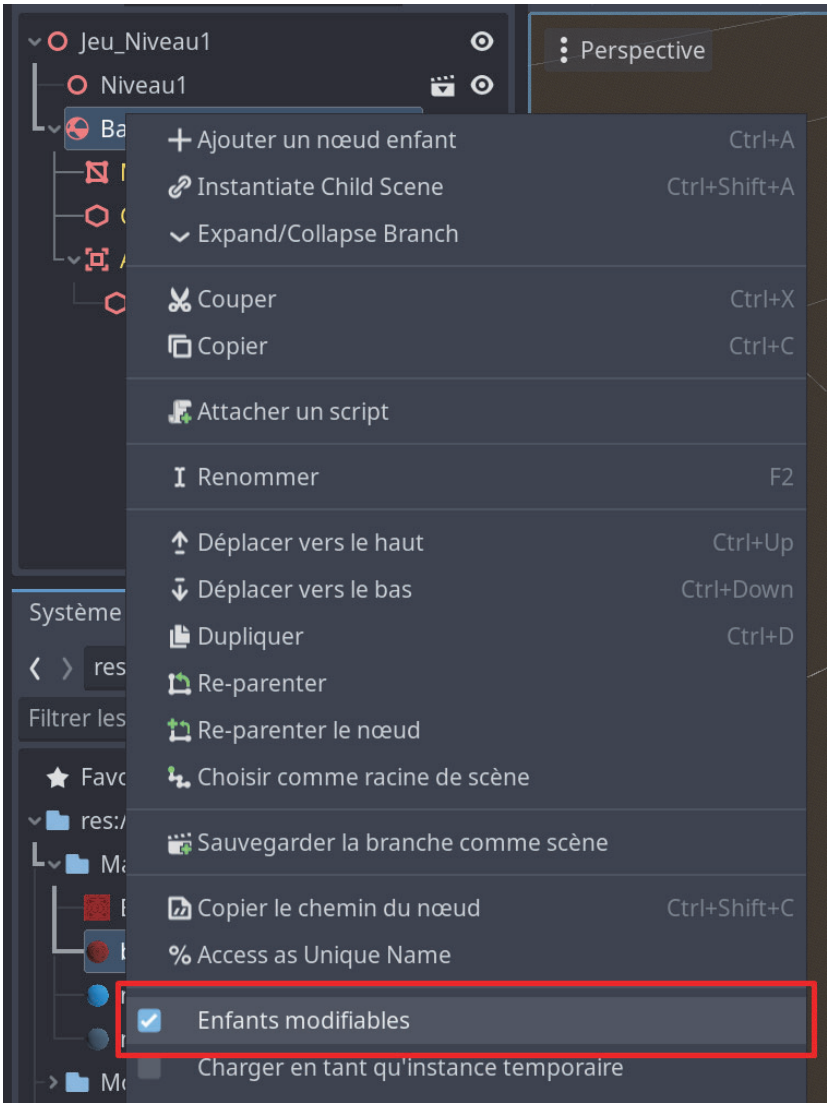
Retournez sur la scène du jeu et instanciez la balle.

Figure 21.15 : *Instanciation de la balle*



Nous avons ajouté la balle à notre jeu ! Il se peut, selon votre modélisation, que la balle soit trop grande par rapport au modèle du niveau. Si c'est le cas, faites un clic droit sur son composant en passant par l'arbre et cochez l'option permettant de rendre les enfants modifiables dans le menu déroulant qui s'affiche. Cela vous permettra de modifier les enfants du nœud.

Figure 21.16 : *Enfants modifiables*

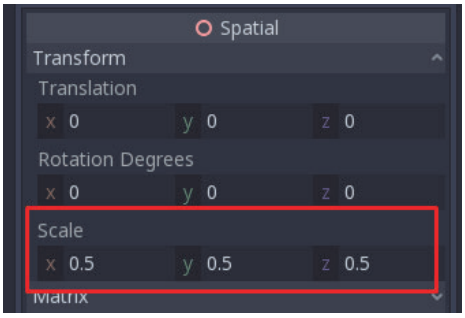


Sélectionnez ensuite tous les enfants de la balle (Maj+clic).

Figure 21.17 : *Sélection multiple*

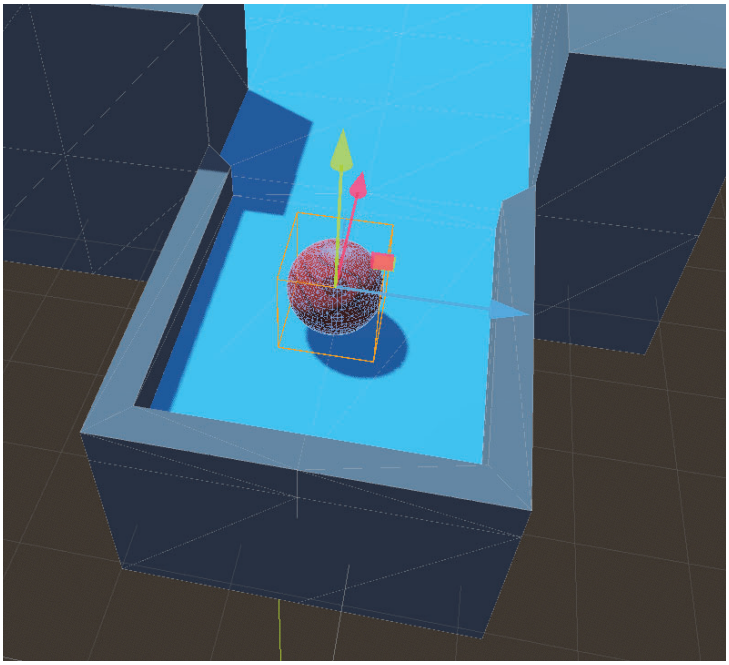
Ensuite, via l'inspecteur, modifiez le Transform afin de réduire la taille des composants. Vous pouvez par exemple spécifier une taille de 0.5 en X, Y et Z pour diviser la taille par deux.

Figure 21.18 : *Modification taille*



La balle aura une taille plus adaptée au niveau :

Figure 21.19 : *Niveau 1 avec la balle*

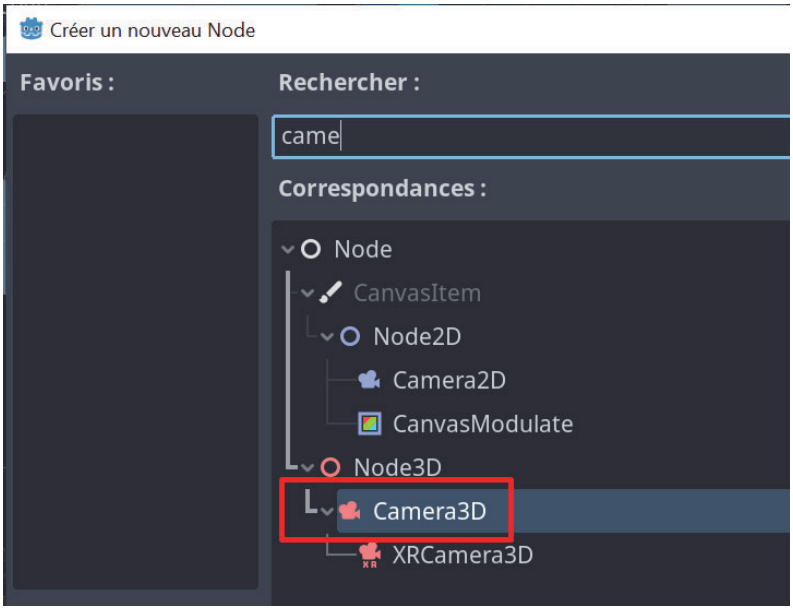


Note › *Encore une fois, tout dépend de votre modélisation. Si les proportions sont correctes dès l'étape de la modélisation 3D alors vous n'aurez pas besoin de modifier la taille de la balle. Mais vous savez maintenant comment procéder si besoin (exemple pour de futurs modèles 3D).*

21.4. Ajout d'une caméra

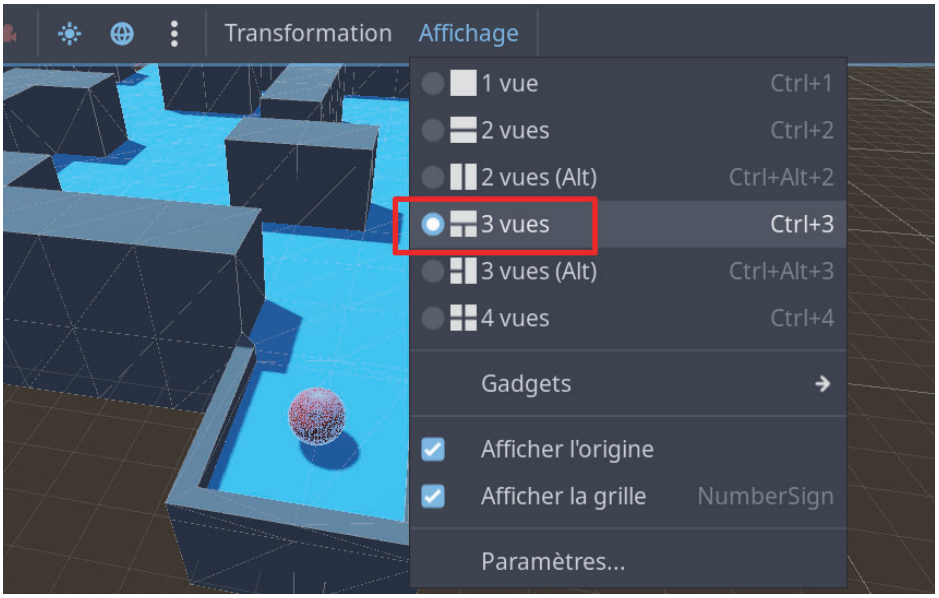
Pour terminer, afin de vérifier que tout fonctionne, ajoutez une caméra.

Figure 21.20 : *Ajout d'une caméra*



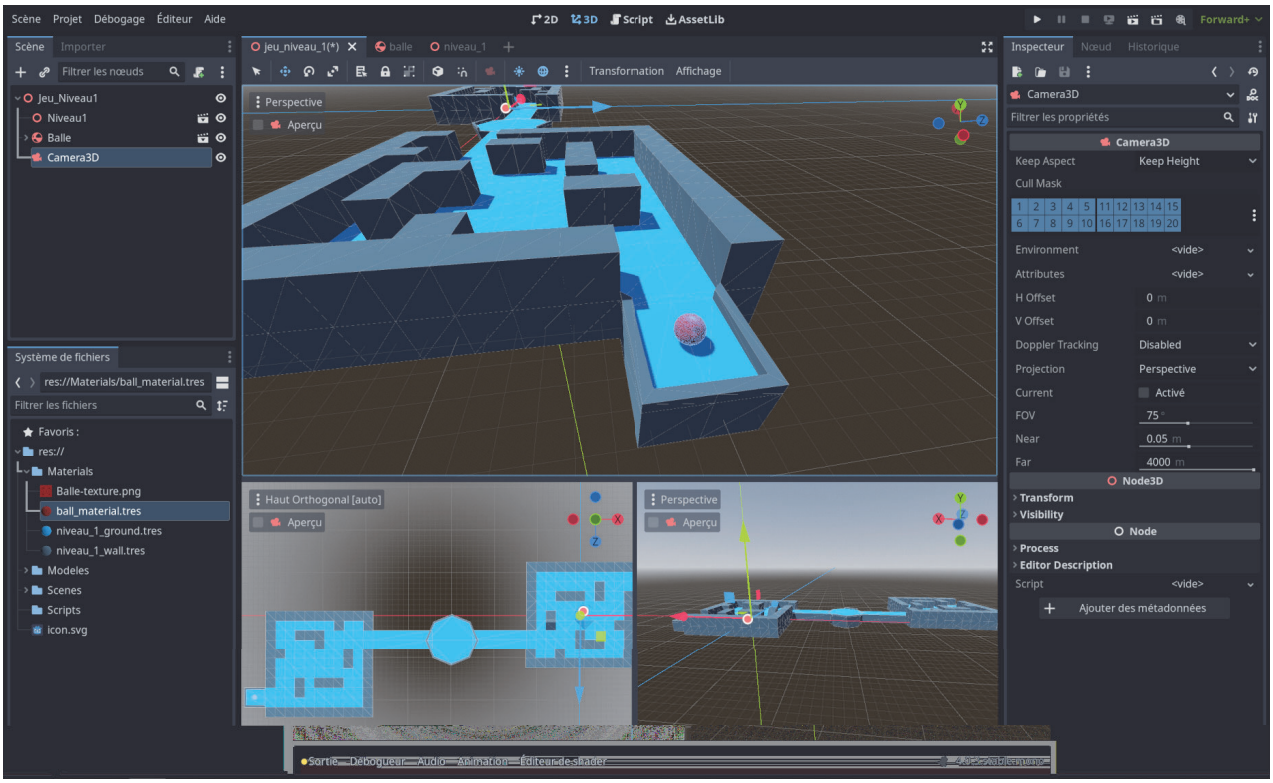
Positionnez la caméra de façon à voir au mieux la balle. Je vous propose de passer en affichage trois vues.

Figure 21.21 : *Affichage trois vues*



Cet affichage vous permettra d'avoir une meilleure vision. L'affichage principal est utile pour conserver la vue de base, l'affichage de gauche peut être utilisé pour avoir une vue de dessus par exemple (touche 7) et l'affichage de droite peut afficher la vue de la caméra en cochant l'option APERÇU.

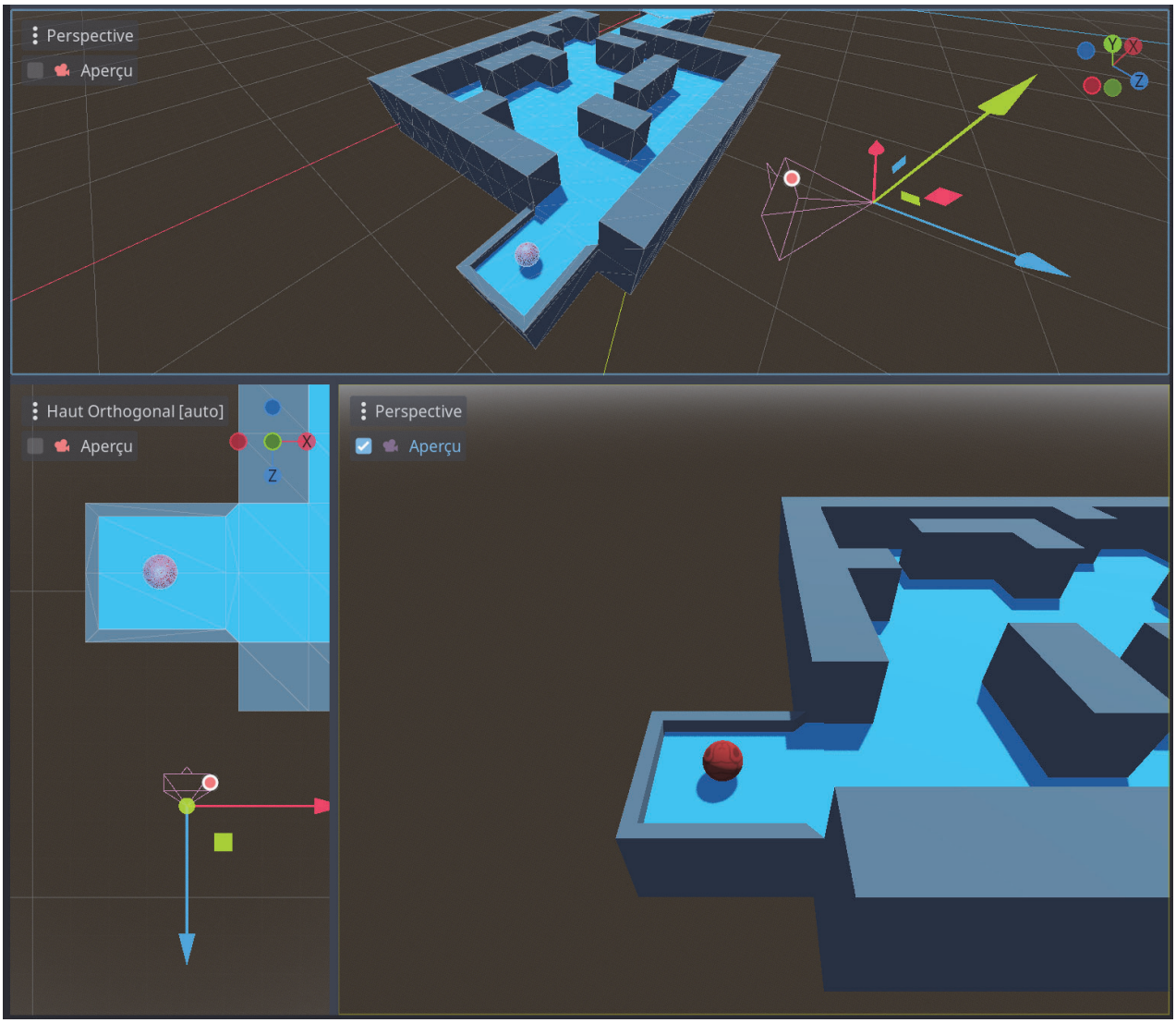
Figure 21.22 : *Triple vue*



Grâce à cette configuration vous pourrez travailler dans de meilleures conditions et placer correctement les objets. Ce que vous pouvez voir en bas à droite est la vue telle qu'aura le joueur final, au travers de la caméra que vous venez d'ajouter.

Essayez de placer la caméra un peu comme dans l'exemple que je vous propose, en vue de côté, une caméra placée au-dessus de la balle un peu inclinée.

Figure 21.23 : *La caméra du jeu*



Pour terminer, testez afin de vérifier que tout fonctionne bien. Enregistrez la scène et appuyez sur F6 pour lancer la scène. Si la balle tombe sur le sol et que les collisions sont prises en compte, tout va bien !En effet, l'important c'est d'avoir la gestion des collisions. Pour le reste, vous ne pouvez rien y faire pour l'instant.

En revanche, si vous avez un souci de collisions, relisez ce chapitre et vérifiez la configuration de vos composants.

Notre scène est configurée. Le visuel n'est pas encore idéal, les couleurs sont sombres et l'éclairage inexistant. Nous améliorerons cela dès le prochain chapitre.

22

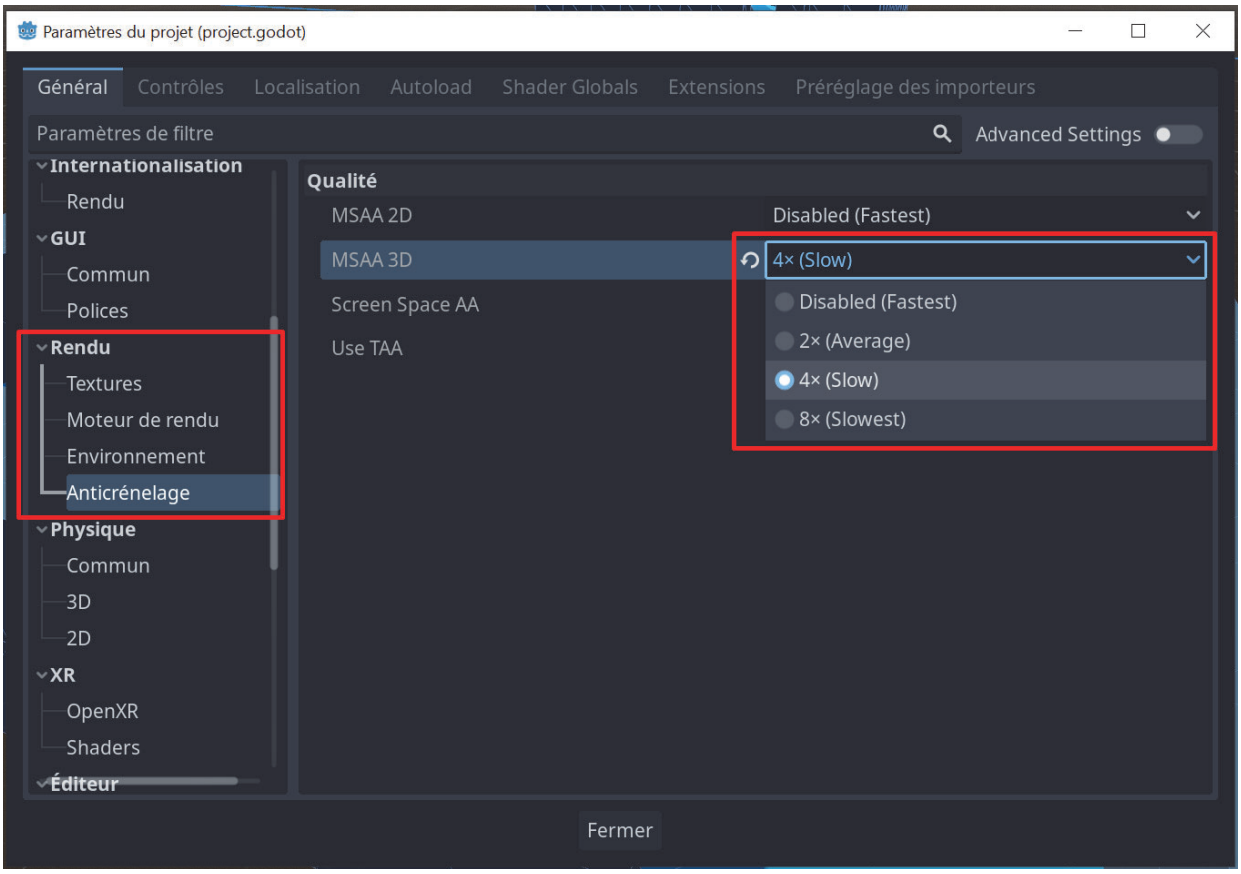
Anticrénelage, éclairage et post-processing

Dans ce chapitre, nous allons voir comment améliorer le rendu de notre scène. Nous configurerons l'éclairage ainsi que la caméra via des effets de post-traitement.

22.1. Anticrénelage

Le premier réglage que nous allons faire, c'est activer l'anticrénelage dans les paramètres du projet. Choisissez une valeur entre 2× ou 8×. Plus la valeur est haute, plus le résultat est bon mais plus les performances sont impactées.

Figure 22.1 : Activation de l'anticrénelage



ANTICRÉNELAGE

L'*anticrénelage* [ou *antialiasing* en anglais] permet de lisser les arêtes des modèles 3D. Voici un exemple concret pour bien comprendre la différence entre une image de base et une image lissée :

Figure 22.2 : *À gauche sans antialiasing et à droite avec*

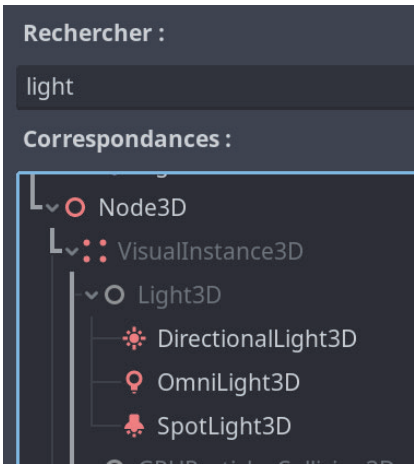
22.2. Éclairage

Dans le jeu vidéo, l'éclairage est très important. Ce sont les lumières qui vont sublimer vos niveaux, en modifier l'ambiance, mettre en évidence certaines zones et guider le joueur. Sans lumière, votre jeu sera plongé dans la nuit. Les joueurs ne verront pas correctement vos modèles 3D ni leurs couleurs. De plus, sans source de lumière il n'y aura pas d'ombres, ce qui rendra votre jeu moins réaliste.

Par défaut Godot (comme beaucoup d'autres moteurs de jeux) met en place un éclairage ambiant très faible permettant de ne pas être dans le noir absolu mais cet éclairage n'est pas suffisant. Nous avons à notre disposition plusieurs types de lumières que nous pouvons utiliser pour mettre en valeur nos niveaux. Si vous recherchez *Light* dans les nœuds disponibles, vous trouverez ces trois principales lumières 3D :

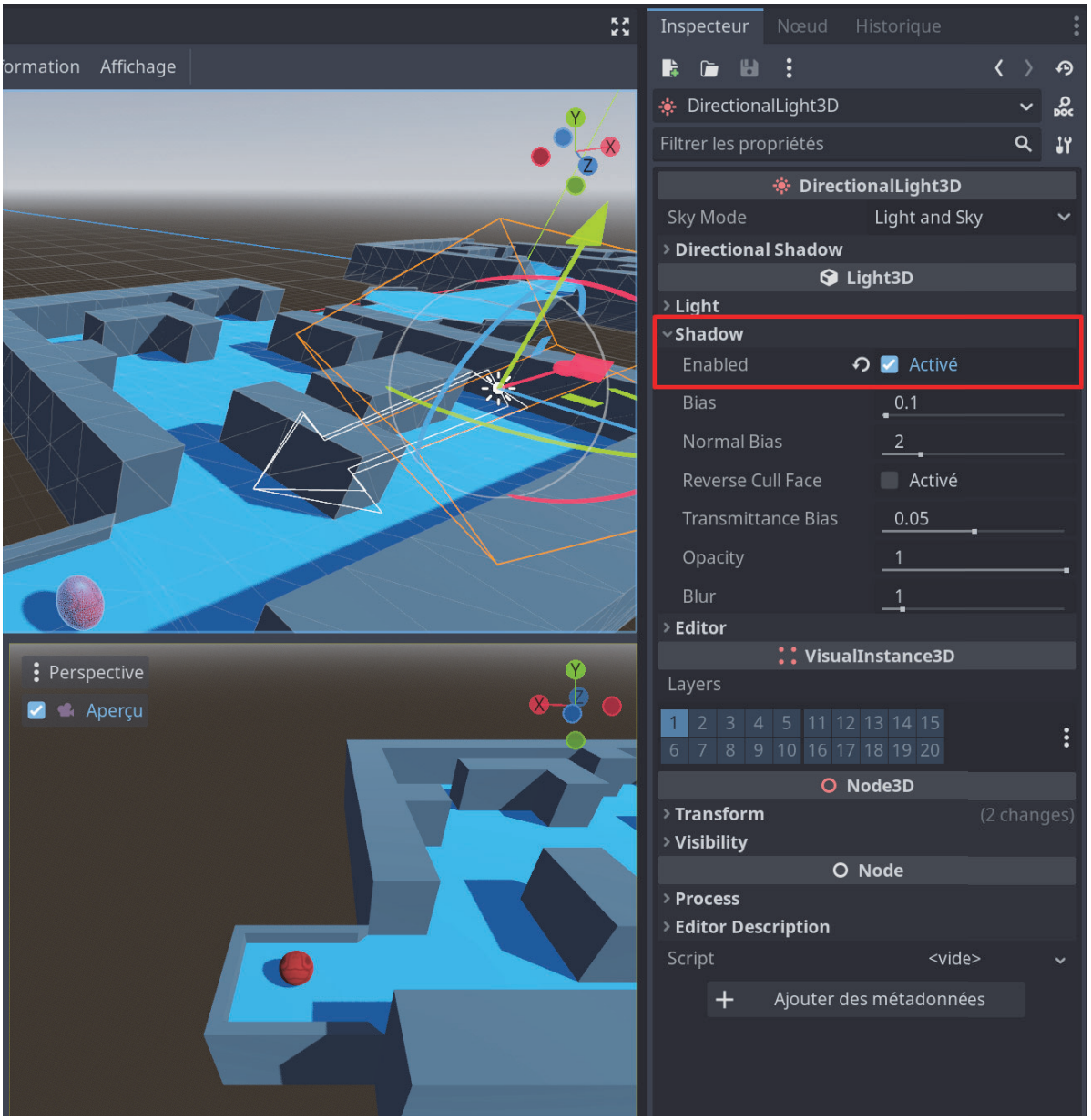
- OmniLight3D, que l'on peut assimiler à un éclairage de type bougie ;
- SpotLight3D, qui est un éclairage type lampe de poche ;
- DirectionalLight3D, que l'on peut comparer à l'éclairage du soleil. C'est une lumière qui éclaire toute la scène à l'infini de façon homogène. Seule la rotation de la lumière influe sur l'éclairage.

Figure 22.3 : *Les différents types de lumières*



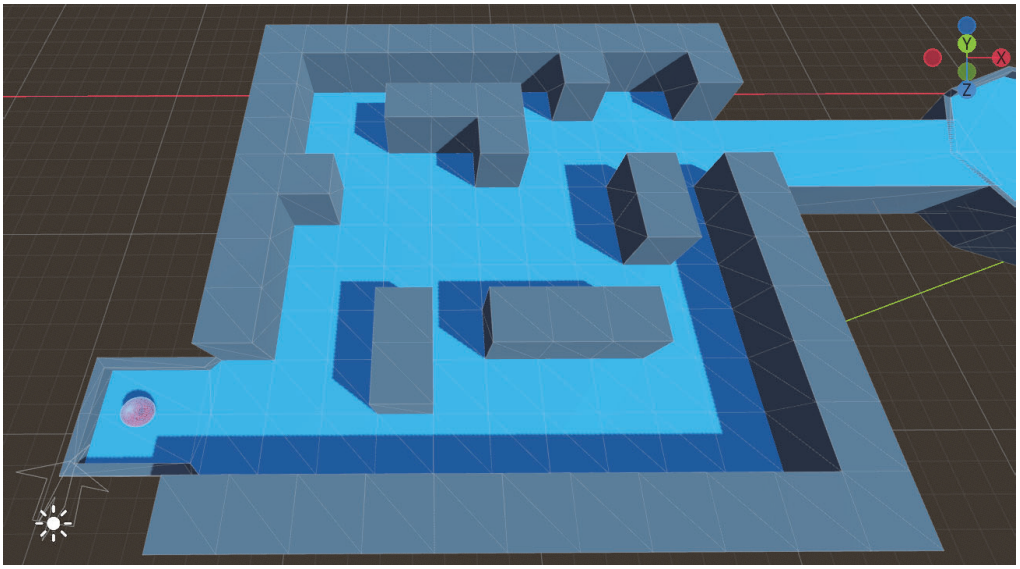
Je vous recommande d'utiliser une lumière directionnelle (DirectionalLight3D) pour l'éclairage global de votre niveau. Placez la lumière où vous le souhaitez (cela n'a aucun impact car la lumière arrive de l'infini), mais orientez-la de sorte à générer des ombres dans le sens que vous voulez. Les ombres doivent être activées dans l'inspecteur.

Figure 22.4 : *Activation des ombres*



Grâce à cette lumière, les couleurs seront beaucoup plus proches de ce que vous avez paramétré dans chaque Material. Vous aurez également des effets de réflexion sur les objets lisses ou brillants (propriétés METALLIC ou ROUGHNESS). Voici le résultat que vous obtenez avec cette lumière directionnelle :

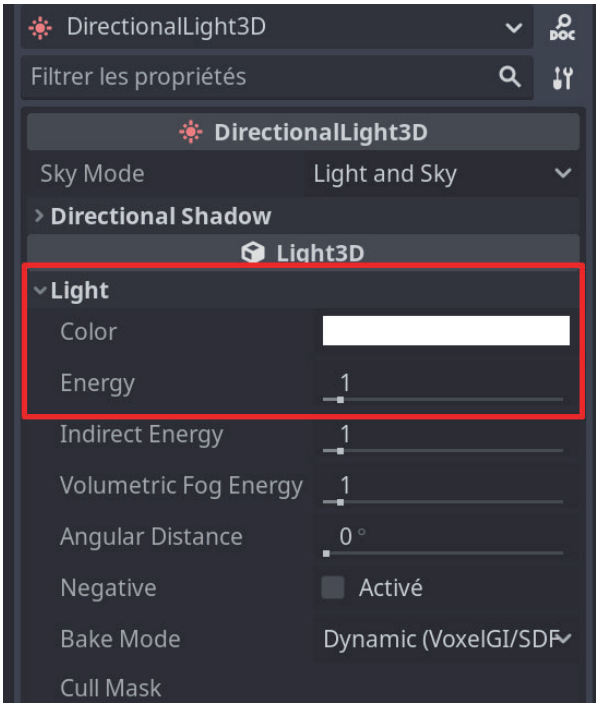
Figure 22.5 : *Lumière directionnelle*



Visuellement le rendu est bien meilleur. Pour ma part, je n'utiliserai qu'une lumière directionnelle pour ce niveau 1. Si vous souhaitez créer des effets et des jeux de lumières, vous pouvez utiliser des SpotLights pour éclairer de façon particulière certains endroits précis pour par exemple mettre en évidence une zone du niveau.

Chaque lumière peut être configurée. Vous pouvez définir la couleur de l'éclairage et son intensité via l'inspecteur.

Figure 22.6 : *Paramètres des lumières*



Enfin, même si je vous conseille de laisser les valeurs par défaut, sachez que vous pouvez modifier la qualité des ombres dans les paramètres du projet. Par défaut les ombres sont de bonne qualité. Vous avez un niveau avec une faible qualité et un autre avec une très haute qualité. La valeur par défaut est le bon compromis qualité/performance.

Figure 22.7 : *Différents niveaux de qualité (Source : docs.godotengine.org)*

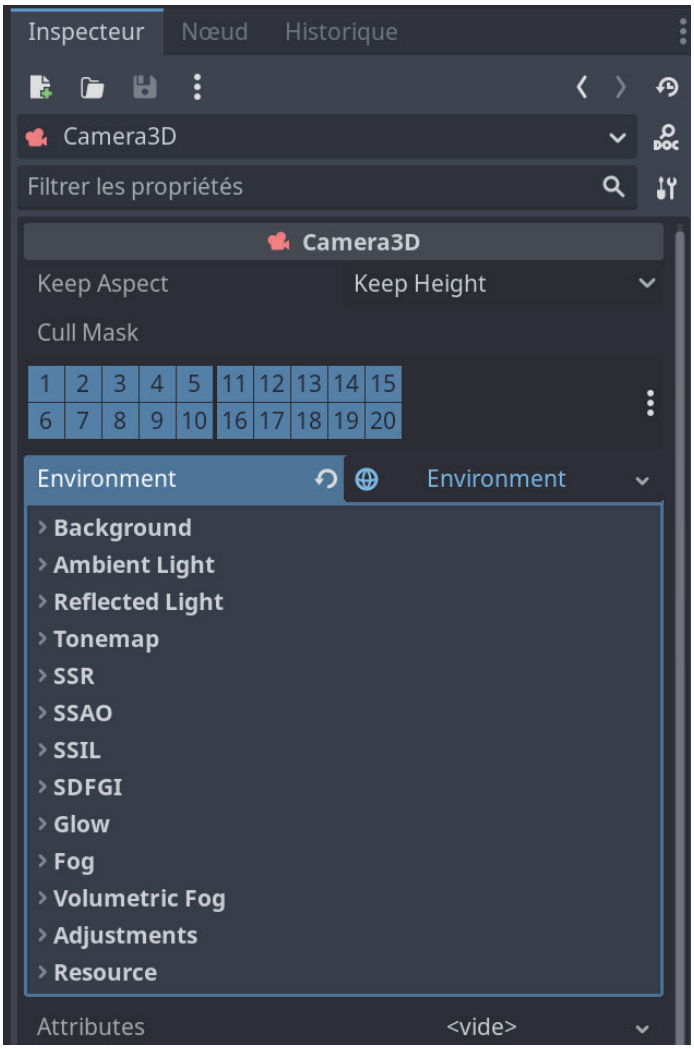
Vous pouvez mettre en place plusieurs lumières si besoin mais évitez de trop en créer car l'éclairage demande beaucoup de ressources. Si vous créez un jeu mobile, essayez de ne garder qu'une lumière directionnelle car la puissance de calcul d'un téléphone est plus faible que celle d'un PC.

22.3. Post-traitement

Le post-traitement (ou *post-processing* en anglais) est un traitement appliqué à l'image 3D après que celle-ci a été calculée. Le principe consiste à effectuer une modification de l'image via des algorithmes afin de mettre en place des effets artistiques plus ou moins complexes. Grâce au post-traitement vous pourrez retravailler l'image, les couleurs, les effets, etc.

Pour activer les effets de post-traitement, vous devez avoir une caméra sur la scène, chose que nous avons déjà. Cliquez sur celle-ci et recherchez dans l'inspecteur la propriété ENVIRONMENT. Créez un nouvel environnement puis cliquez sur celui-ci.

Figure 22.8 : *Création d'un environnement*

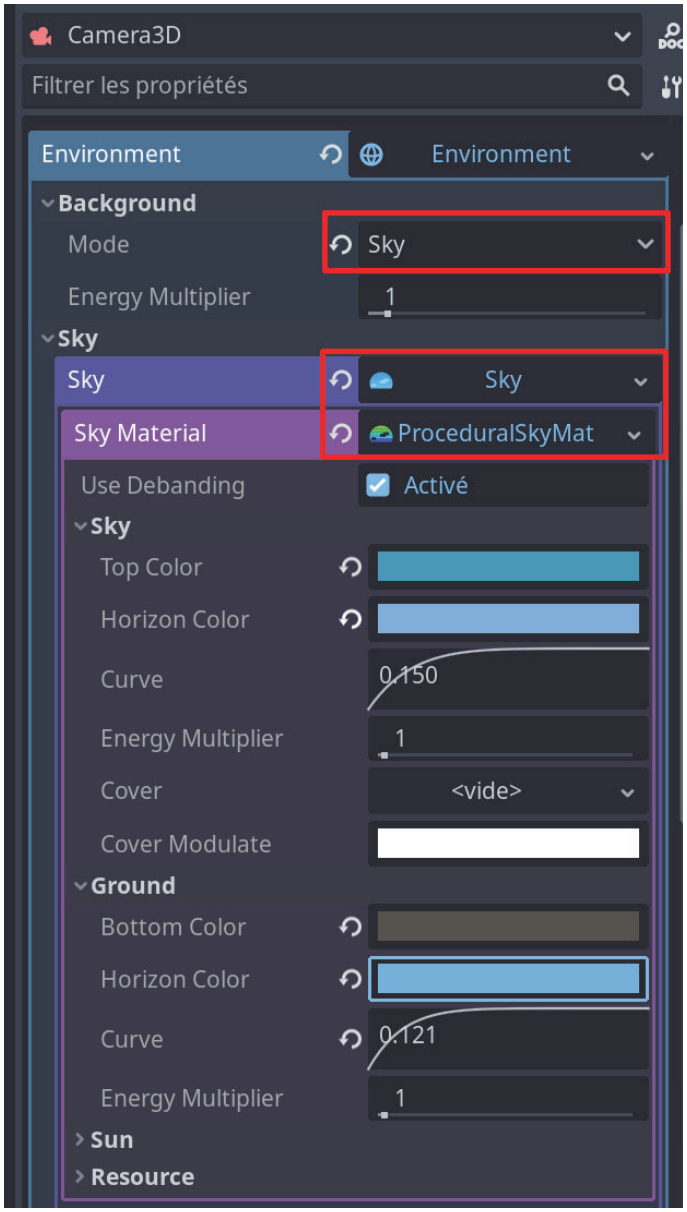


Dès lors que vous avez préparé cette propriété ENVIRONMENT, vous pouvez activer certains effets. Nous allons les découvrir un à un et activer ceux dont nous aurons besoin.

BACKGROUND

La propriété BACKGROUND vous permet de paramétrer l'arrière-plan. Vous pouvez mettre en place une couleur unie ou une skybox afin de créer un ciel et l'horizon. Choisissez `Sky` pour mettre en place un ciel et `ProceduralSky` pour créer un ciel procédural (et non un ciel basé sur des photos). Le ciel procédural est idéal pour créer par exemple un cycle jour/nuit ou simplement pour avoir une couleur avec un beau dégradé. Voilà quelques réglages que vous pouvez ajuster pour trouver la couleur qu'il vous faut :

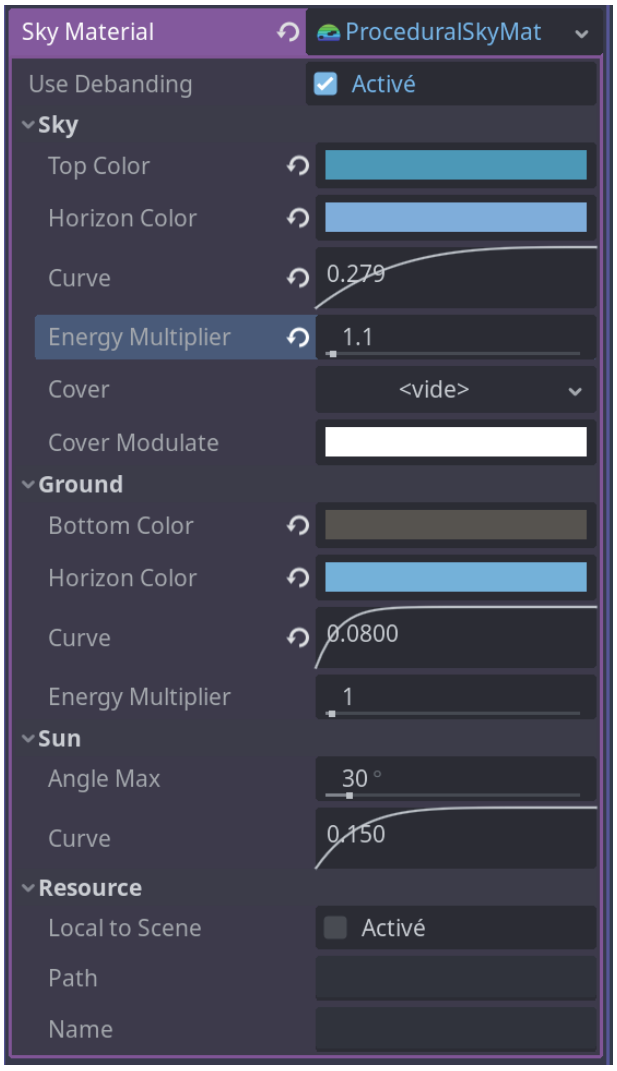
Figure 22.9 : *Paramètres du ciel*



En plus de la lumière directionnelle, le ciel a un léger impact sur l'éclairage de votre scène. Pour en renforcer un peu l'effet, j'ai augmenté légèrement son énergie.

Si vous cliquez sur PROCEDURALSKY, vous aurez accès à davantage de propriétés vous permettant de modifier les couleurs, le dégradé, le soleil, la courbe de l'horizon etc.

Figure 22.10 : *Les options supplémentaires du ciel*



AMBIENTLIGHT

La propriété AMBIENTLIGHT permet de configurer l'éclairage ambiant. Vous pouvez choisir la couleur de cet éclairage ainsi que son intensité. Je vous propose de laisser les valeurs par défaut pour notre projet.

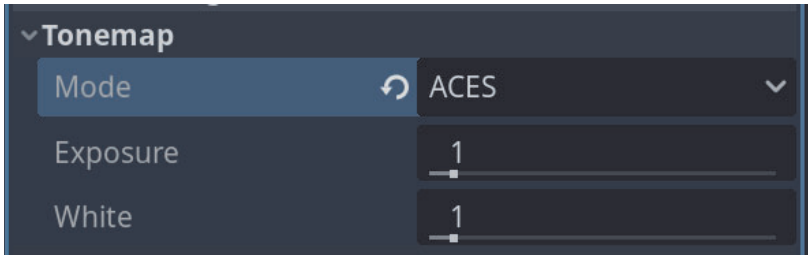
FOG

La propriété FOG permet de mettre en place un brouillard. C'est une pratique courante dans le jeu vidéo pour masquer l'arrière-plan et ne pas avoir à le modéliser, et ainsi limiter les calculs que doit faire la carte graphique. Vous pouvez définir la couleur du brouillard, sa densité, son épaisseur par rapport à la distance de la caméra, etc. Pour ma part, je ne mettrai pas de brouillard dans ce projet.

TONEMAP

TONEMAP est une propriété très intéressante qui va vous permettre de retravailler les couleurs de votre scène. Je vous propose d'essayer le filtre ACES qui met en place un bon contraste.

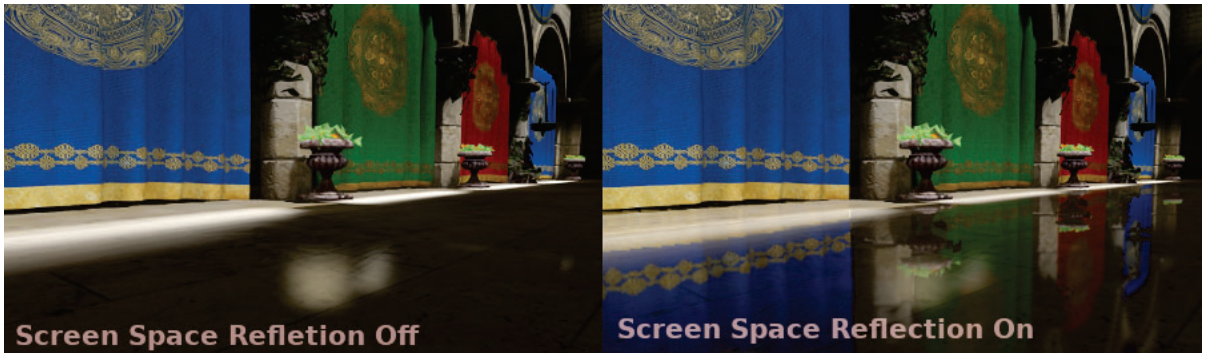
Figure 22.11 : *Tonemap Aces*



SSR

SSR ou SCREEN SPACE REFLECTION est un effet permettant de créer des reflets de l'environnement sur les surfaces lisses réfléchissantes ou les flaques d'eau au sol. Dans notre cas, l'effet ne sera pas trop visible car notre décor est basique mais voici à quoi s'attendre avec un décor un peu plus garni.

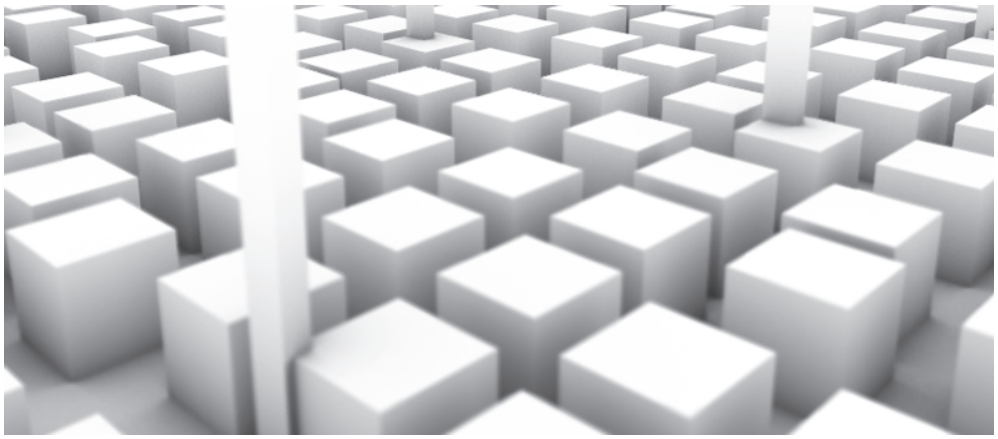
Figure 22.12 : *Sans/Avec SS Reflection (Source : docs.godotengine.org)*



SSAO

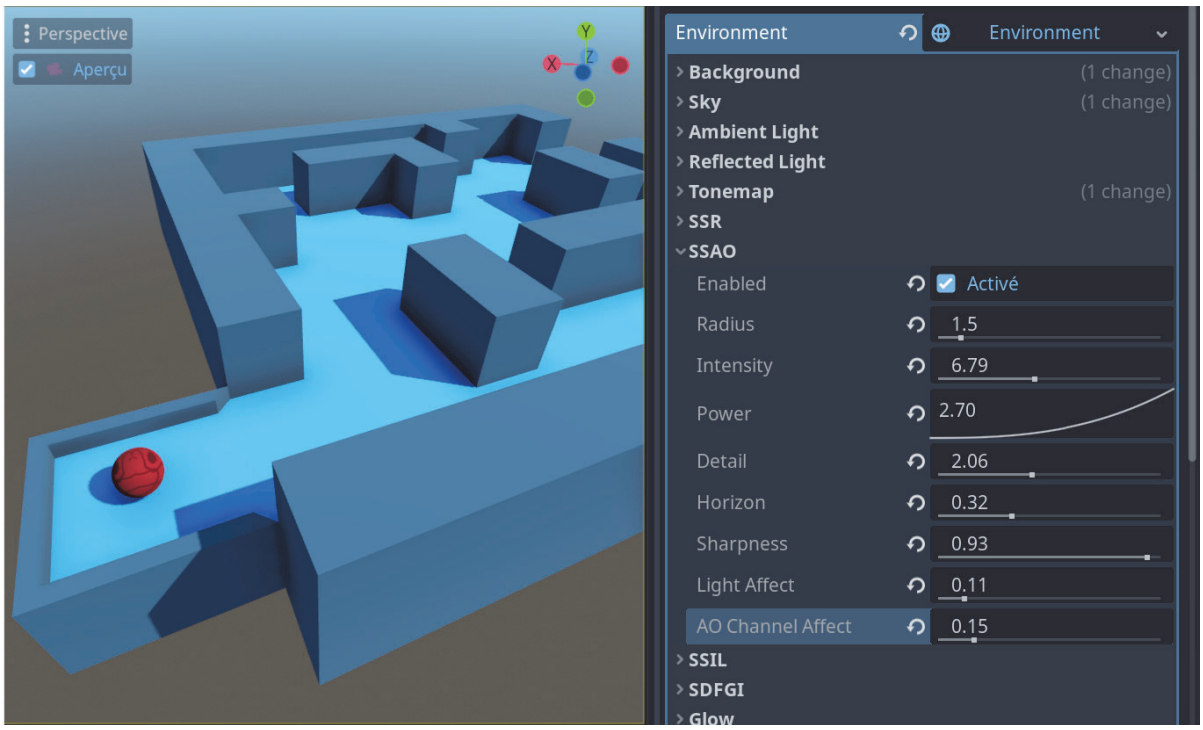
SSAO (Screen Space Ambiant Occlusion) est un effet qui applique des ombres localisées sur les arêtes, les coins et les renfoncements de vos modèles 3D. Voici un petit exemple de scène composée de simples cubes avec de l'Ambiant Occlusion.

Figure 22.13 : *SSAO (Source : Wikimédia)*



Vous pouvez activer cet effet pour bien délimiter la géométrie, intensifier les ombres et créer un rendu sympa. Ajustez l'intensité de l'effet et les paramètres selon votre goût. Voici mes réglages et le résultat :

Figure 22.14 : *SSAO*



GLOW

La propriété GLOW permet de créer un effet de lueur ou d'éclat sur les objets brillants ou réfléchissants. Cet effet est, par exemple, très intéressant sur le pare-brise des voitures. La Figure 22.15 vous en donne un aperçu.

Figure 22.15 : *Glow*

ADJUSTMENTS

La propriété ADJUSTMENTS permet, comme le nom l'indique, de faire des ajustements sur l'image, notamment la luminosité, le contraste ou la saturation. Pour notre exemple, nous pouvons utiliser cet effet pour renforcer les couleurs du niveau avec les paramètres suivants :

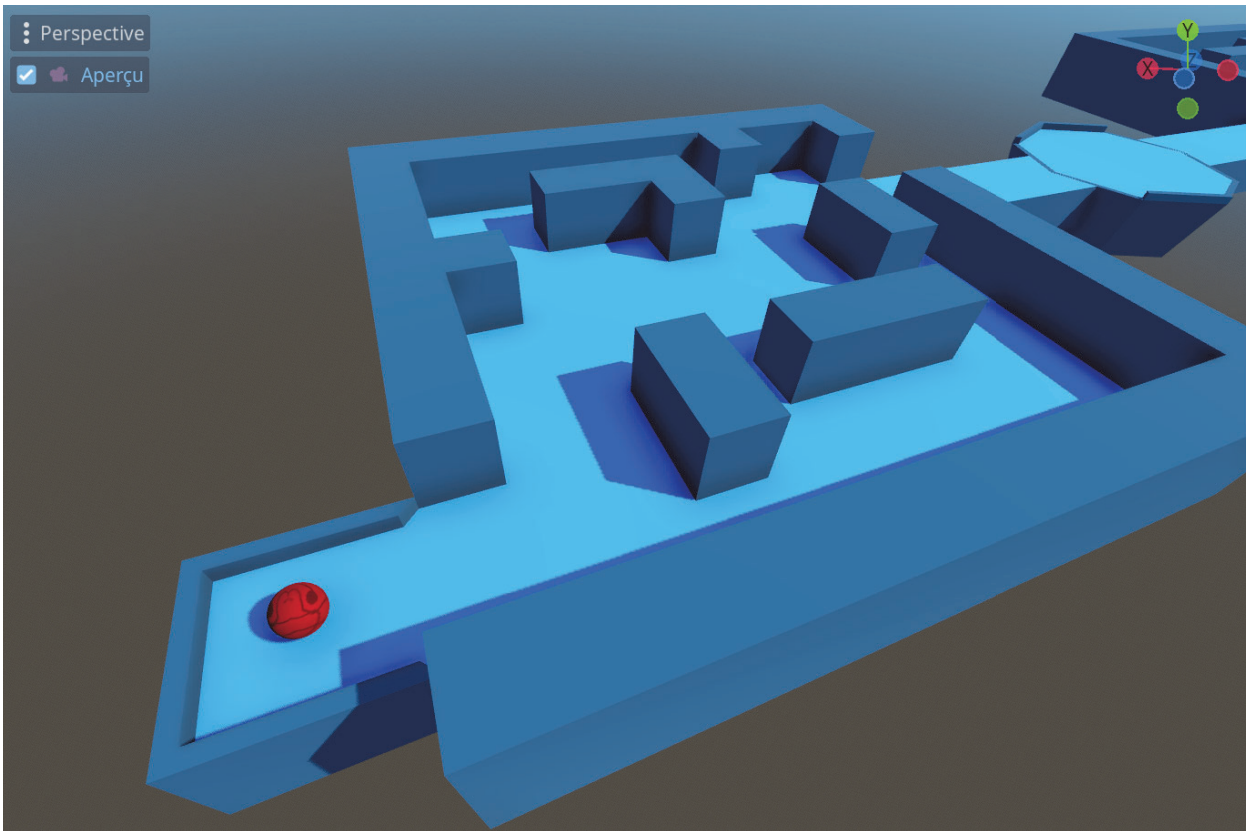
Figure 22.16 : *Adjustments*

SSIL & SDFIG

SSIL & SDFIG sont des propriétés qui ont été ajoutées récemment dans Godot. Elles permettent d'obtenir un rendu plus précis et plus réaliste au niveau de l'éclairage.

La Figure 22.17 vous montre le rendu final amélioré grâce au post-traitement.

Figure 22.17 : *Rendu final*



Le visuel est plus détaillé et les couleurs ressortent mieux. L'avantage de cette technique c'est que vous pouvez modifier l'ambiance visuelle de votre jeu sans avoir besoin de toucher aux textures car ce sont des algorithmes qui vont ajuster le visuel de base.

Vous avez donc vu comment améliorer grandement le rendu d'une scène en quelques clics sans avoir à toucher aux modèles 3D ni aux textures. Vous pouvez jouer avec ces paramètres pour trouver le style qui vous ressemble. Dans le chapitre suivant, nous passerons à la programmation.

23

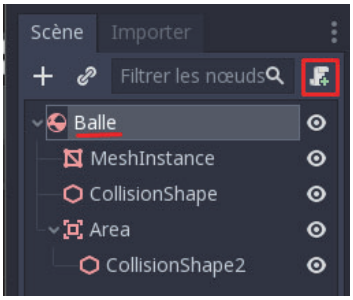
Déplacement de la balle

Notre environnement 3D est configuré et ne demande plus qu'à être exploré. C'est ce que nous allons faire en donnant la possibilité au joueur de déplacer la balle avec son clavier afin de naviguer dans ce premier niveau. Nous allons commencer par créer un script basique permettant d'utiliser les flèches directionnelles pour faire rouler la balle. Nous aurons également besoin de paramétrer la caméra afin que celle-ci suive la balle pendant son déplacement.

23.1. Script de la balle

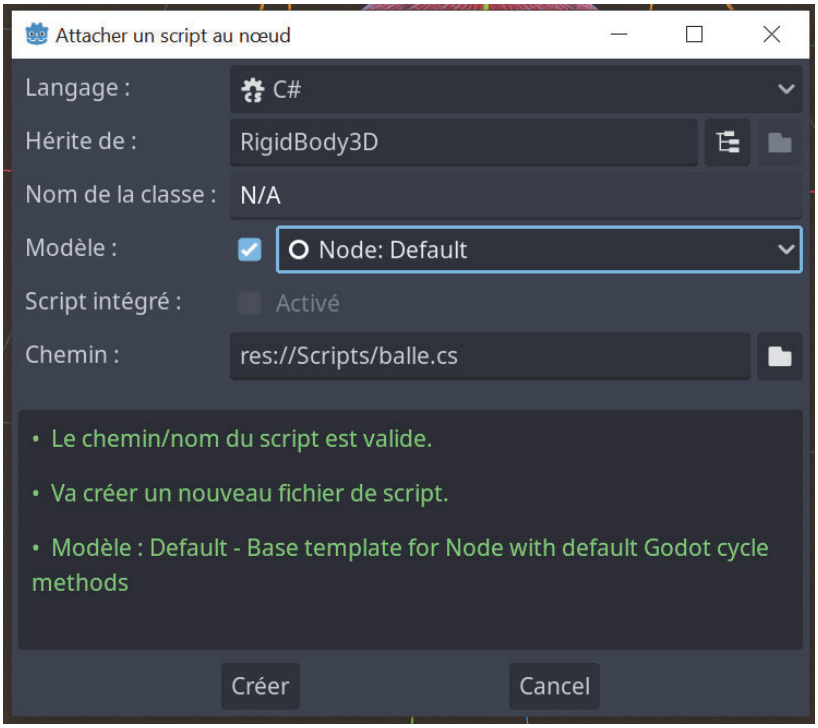
Retournez sur la scène de la balle. C'est là que nous allons créer notre script. Cliquez sur le nœud principal (c'est-à-dire le RigidBody `Balle`) puis sur l'icône Script (en haut à droite du panneau Scène) afin d'ajouter un script .

Figure 23.1 : Ajouter un script



Choisissez le modèle C# par défaut et enregistrez le script dans le dossier des scripts.

Figure 23.2 : *Attacher le script*



Lorsque vous validez, le script s'ouvre. Commencez par créer une variable `force` avec une valeur de 1.

```
[Export] public float force = 1f;
```

L'annotation `[export]` permet d'afficher cette variable dans l'inspecteur et de modifier sa valeur à la volée sans passer par le script.

Nous allons à présent créer une fonction `GetAxis` qui nous permettra de savoir quel axe (vertical ou horizontal) est utilisé par le joueur. Selon l'axe utilisé, nous retournerons une valeur positive ou négative en X ou en Y. Pour identifier l'axe puis la touche qui a été pressée, nous utiliserons des conditions.

```
public int GetAxis(string axis)
{
  // Si on effectue un mouvement horizontal
  if (axis == "horizontal")
  {
    // Flèche de gauche
    if (Input.IsActionPressed("ui_left"))
      return -1;
    // Droite
    else if (Input.IsActionPressed("ui_right"))
      return 1;
  }
```

```
    // Si vertical
    else if (axis == "vertical")
    {
      // Haut
      if (Input.IsActionPressed("ui_up"))
        return -1;
      // Bas
      else if (Input.IsActionPressed("ui_down"))
        return 1;
    }
    return 0;
  }
```

Nous pourrons désormais utiliser cette fonction `GetAxis` pour retourner le multiplicateur adéquat. Par défaut, si l'utilisateur n'appuie sur aucune touche Flèche, nous retournons zéro afin que le mouvement soit nul et donc que la balle ne bouge pas.

Nous allons maintenant utiliser la fonction `_PhysicsProcess` qui nous sera très utile pour appliquer le mouvement à la balle car ce mouvement doit être continu et donc se produire 60 fois par seconde si le jeu tourne en 60 images par seconde. Nous utilisons `_PhysicsProcess` au lieu de `_Process` car elle est plus adaptée aux composants physiques comme le RigidBody. Elle nous permettra, grâce à la propriété `delta`, de calibrer la vitesse de déplacement indépendamment de la puissance de l'ordinateur faisant tourner le jeu.

Créez donc cette fonction :

```
public override void _PhysicsProcess(double delta)
```

Puis créez deux variables permettant de stocker le déplacement en X et le déplacement en Y en appelant notre fonction `GetAxis` :

```
// Mouvement en X
int xSpeed = GetAxis("horizontal");
// Mouvement en Y
int ySpeed = GetAxis("vertical");
```

Créez ensuite une variable pour stocker la direction qui sera un `Vector3` composé des mouvements X et Y :

```
var dir = new Vector3(xSpeed, 0, ySpeed).Normalized();
```

J'applique la méthode `Normalized` afin de normaliser le mouvement. Cela permet de calibrer la vitesse. Si vous ne le faites pas, la balle ira plus vite lorsque le joueur se déplacera simultanément en X et en Y.

Créez ensuite une autre variable qui stockera la vitesse du mouvement. Cette vitesse est égale à la force multipliée par `delta` pour qu'elle soit identique sur tous les ordinateurs.

```
float magnitude = force * (float)delta;
```

Enfin, appelez la fonction `AddImpulse` que nous n'avons pas encore créée. Elle aura pour rôle d'appliquer la force donnée à la balle :

```
AddImpulse(magnitude * dir);
```

Voici le code complet de notre fonction `_PhysicsProcess` :

```
// Tourne en boucle
public override void _PhysicsProcess(double delta)
{
  // Mouvement en X
  int xSpeed = GetAxis("horizontal");
  // Mouvement en Y
  int ySpeed = GetAxis("vertical");
  // Vecteur 3 de direction
  var dir = new Vector3(xSpeed, 0, ySpeed).Normalized();
  // Pour la vitesse de mouvement
  float magnitude = force * (float)delta;
  // On applique la force de propulsion dans la direction * la force
  AddImpulse(magnitude * dir);
}
```

Nous devons maintenant créer la fonction `AddImpulse`. Cette fonction prend en paramètre la force `f` et applique une impulsion à la balle suivant cette force :

```
public void AddImpulse(Vector3 f)
{
  // Si une force existe en X ou Y
  if (f[0] != 0 || f[2] != 0)
  {
  // On ajoute une impulsion à la balle
  ApplyCentralImpulse (f);
  }
}
```

Ici, nous testons si `f` (la force) a une valeur non nulle puis nous utilisons `ApplyCentralImpulse`, fonction permettant d'appliquer l'impulsion. Le paramètre correspond au vecteur de déplacement.

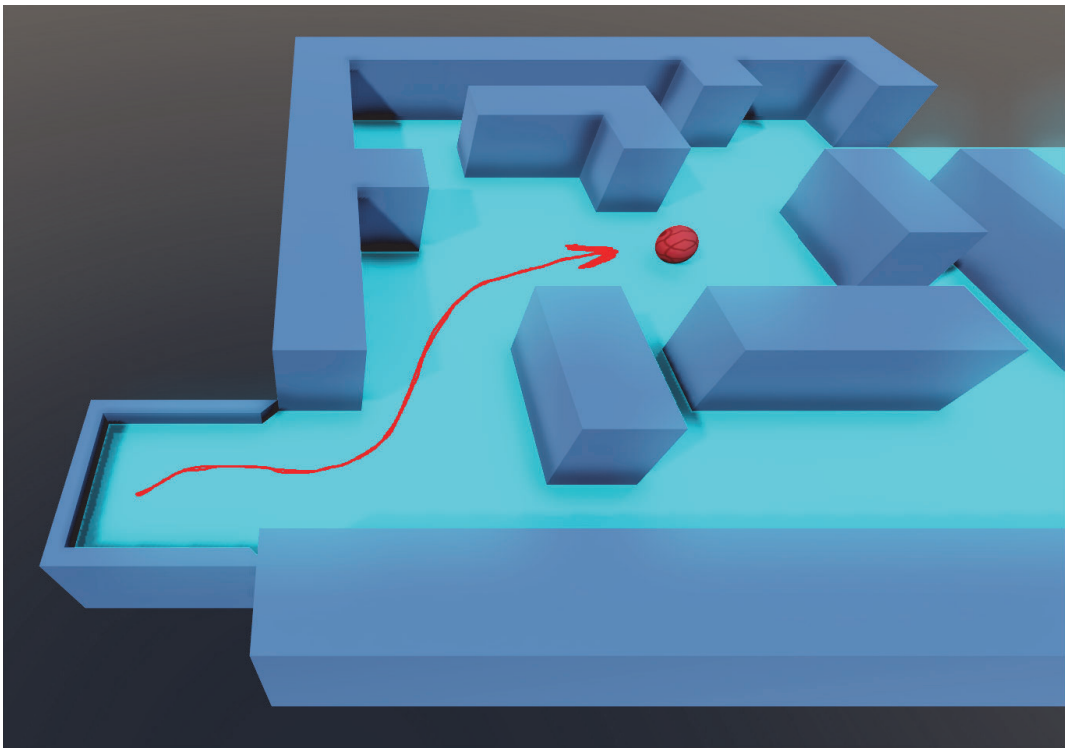
Le code complet pour ce script est le suivant :

```
using Godot;
```

```
using System;

public partial class balle : RigidBody3D
{
  [Export] public float force = 1f;

  public int GetAxis(string axis)
  {
    // Si on effectue un mouvement horizontal
    if (axis == "horizontal")
    {
      // Flèche de gauche
      if (Input.IsActionPressed("ui_left"))
        return -1;
      // Droite
      else if (Input.IsActionPressed("ui_right"))
        return 1;
    }
    // Si vertical
    else if (axis == "vertical")
    {
      // Haut
      if (Input.IsActionPressed("ui_up"))
        return -1;
      // Bas
      else if (Input.IsActionPressed("ui_down"))
        return 1;
    }
    return 0;
  }

  public override void _PhysicsProcess(double delta)
  {
    // Mouvement en X
    int xSpeed = GetAxis("horizontal");
    // Mouvement en Y
    int ySpeed = GetAxis("vertical");
    // Vecteur 3 de direction
    var dir = new Vector3(xSpeed, 0, ySpeed).Normalized();
    // Pour la vitesse de mouvement
    float magnitude = force * (float)delta;
    // On applique la force de propulsion dans la direction * la force
    AddImpulse(magnitude * dir);
  }

  public void AddImpulse(Vector3 f)
  {
    // Si une force existe en X ou Y
    if (f[0] != 0 || f[2] != 0)
    {
      // On ajoute une impulsion à la balle
      ApplyCentralImpulse (f);
    }
  }
}
```

Vous pouvez enregistrer le script et la scène de la balle et vérifier que tout fonctionne. Pensez à ajuster la valeur de la variable FORCE afin de trouver une vitesse convenable. Dans mon cas, une valeur de 10 semble correcte. Retournez sur votre jeu puis lancez-le en appuyant sur F6. Utilisez les flèches du clavier pour vous déplacer.

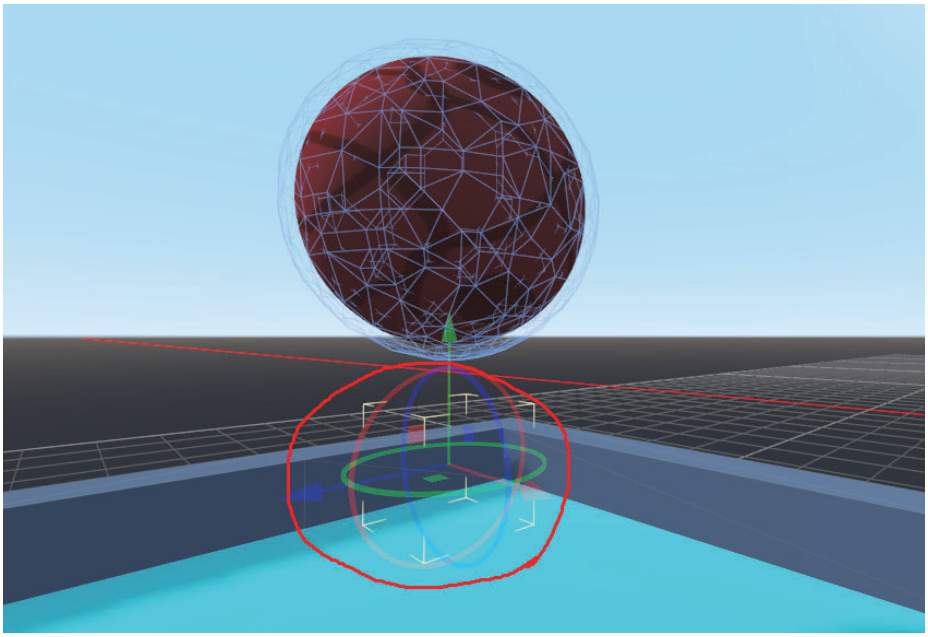
Figure 23.3 : *La balle se déplace*



Si tout va bien, la balle devrait se déplacer. Si cela ne fonctionne pas correctement, voici quelques erreurs possibles :

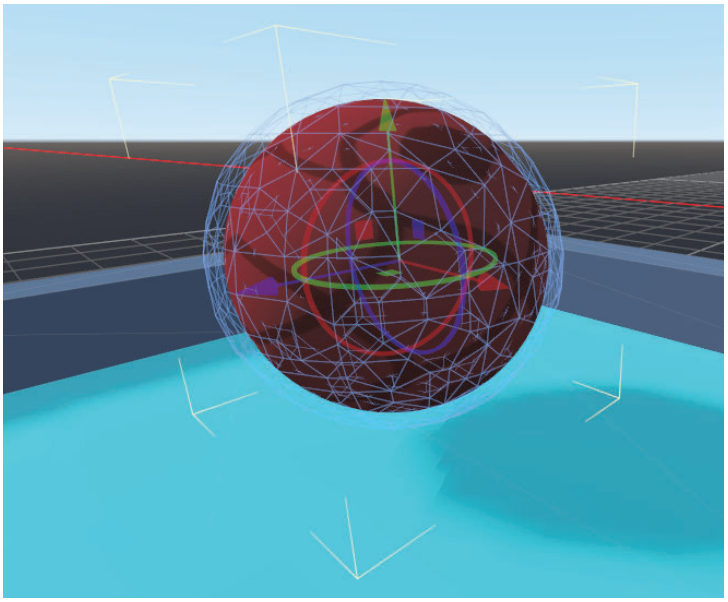
- Une erreur dans le script : vérifiez que vous n'avez pas fait d'erreur dans l'écriture du script, comparez-le à mon script fourni avec les sources du projet fournies avec ce livre.
- Une erreur dans les axes : il se peut que votre niveau ou votre caméra soient inversés. Du coup, les flèches du clavier sont inversées. Si tel est le cas, rien de plus simple, inversez les valeurs dans la fonction `GetAxis` afin de remplacer les 1 par des −1 et inversement.
- Une erreur sur la balle : vous avez peut-être déplacé malencontreusement des éléments enfants de la balle au lieu de déplacer le nœud parent. Votre centre de gravité doit être au centre de la balle. Voici un exemple d'erreur :

Figure 23.4 : *Une erreur de configuration*



Sur cette image, on voit que le centre de gravité est décalé car j'ai déplacé les enfants plutôt que le nœud parent. Vous devriez avoir quelque chose comme sur la Figure 23.5.

Figure 23.5 : *Balle correctement configurée*



Une autre erreur pourrait être due à de mauvais objets de collision (convexe ou concave). Pensez à comparer votre projet au mien en cas de comportement indésirable.

23.2. Script de la caméra

Nous allons maintenant nous atteler au script de la caméra qui sera beaucoup plus simple. Sur la scène de votre niveau 1, ajoutez un script à la caméra. Dans ce script, nous allons avoir besoin d'une variable qui stockera le décalage (`offset`) de la caméra par rapport à la balle et une variable pour indiquer quel nœud doit être suivi par la caméra. Pour stocker la balle dans sa variable, nous passerons par la fonction `_Ready` :

```
[Export] public Vector3 cameraOffset;
public RigidBody3D target;

public override void _Ready()
{
  target = (RigidBody3D)GetParent().GetNode("Balle");
}
```

Dans la fonction `_Ready` qui se lance au début du jeu, récupérez la position de la balle et celle de la caméra via la propriété `Position` de ces objets puis calculez le décalage (`offset`) :

```
public override void _Ready()
{
  target = (RigidBody3D)GetParent().GetNode("Balle");

  var playerPosition = target.Position;
  var cameraPosition = Position;
  cameraOffset = cameraPosition - playerPosition;
}
```

Enfin, dans la fonction `_Process`, récupérez la position de la balle et modifiez celle de la caméra selon la première. Utilisez la variable `cameraOffset` afin de toujours conserver le même décalage de la caméra par rapport à la balle :

```
public override void _Process(double delta)
{
  var playerPosition = target.Position;
  Position = playerPosition + cameraOffset;
}
```

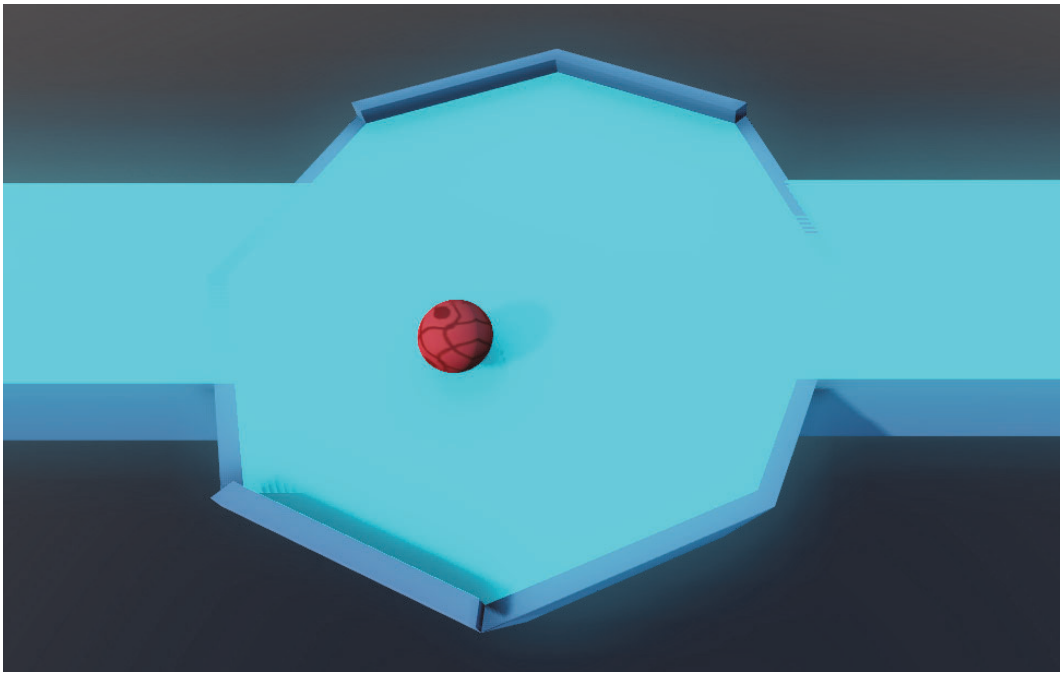
Voici le code complet :

```
using Godot;
using System;

public partial class Camera3D : Godot.Camera3D
```

```
{
  [Export] public Vector3 cameraOffset;
  public RigidBody3D target;

  public override void _Ready()
  {
    target = (RigidBody3D)GetParent().GetNode("Balle");

    var playerPosition = target.Position;
    var cameraPosition = Position;
    cameraOffset = cameraPosition - playerPosition;
  }

  public override void _Process(double delta)
  {
    var playerPosition = target.Position;
    // On positionne la caméra par rapport à la balle
    Position = playerPosition + cameraOffset;
  }
}
```

Enregistrez et testez votre projet.

Figure 23.6 : *La caméra suit la balle*



Vous pouvez maintenant vous déplacer dans le niveau comme bon vous semble et la caméra n'en perdra pas une miette !

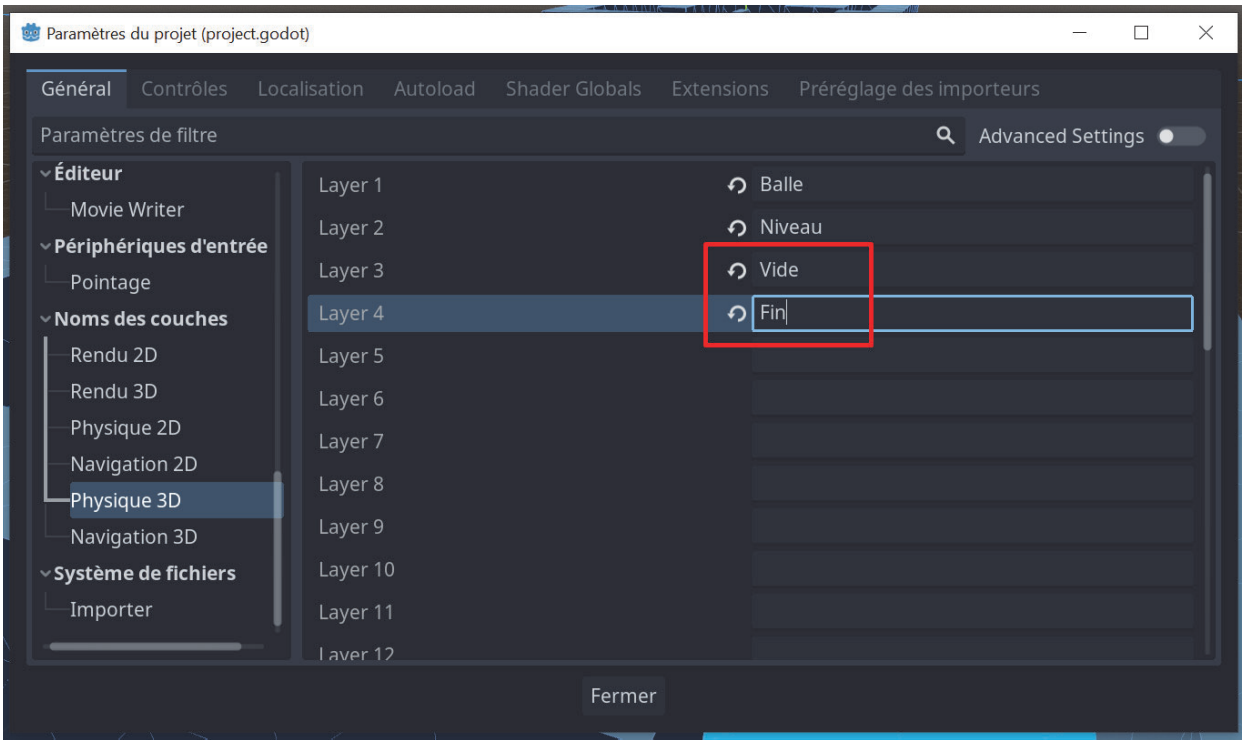
24

Déclenchement de la fin du niveau

Pour le moment, nous pouvons simplement rouler sur le niveau. Il n'est pas possible de terminer celui-ci car nous n'avons pas programmé de déclencheur de fin. De plus, si la balle tombe dans le vide, rien ne se passe et le niveau ne se relance pas. Dans ce chapitre, nous allons mettre en place deux déclencheurs : l'un pour terminer le niveau et l'autre pour le relancer si le joueur tombe.

Pour commencer, nous ajoutons deux calques dans les propriétés du projet afin d'être en mesure de détecter une collision avec le vide (chute dans le vide) ou la fin du niveau :

Figure 24.1 : *Ajout des calques*



Voyons maintenant comment mettre en place les déclencheurs.

24.1. Détecter la fin du niveau

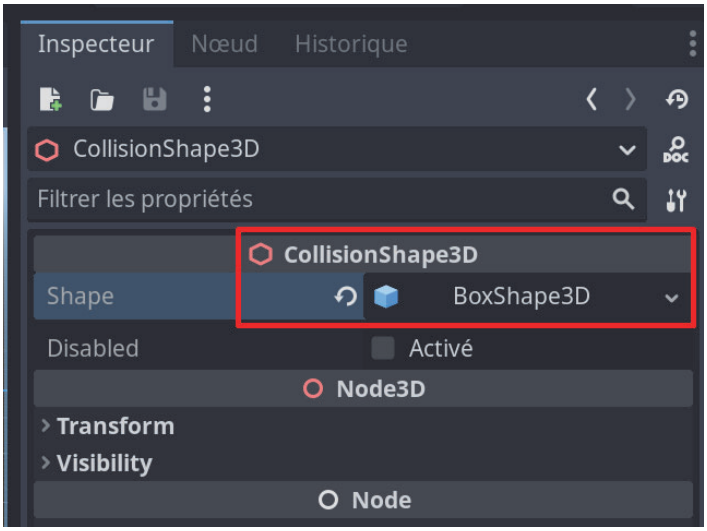
Commençons par coder la fin du niveau. Le but du jeu est d'atteindre l'extrémité du niveau. Nous allons donc placer un objet de collision invisible (`Trigger/Collider/Déclencheur`) afin de détecter si la balle l'atteint.

Nous allons faire quelque chose de simple et rapide à mettre en place. Nous allons créer un Area3D suivi d'un CollisionShape3D pour pouvoir utiliser la fonction `on_Area_entered` de la balle sur cet Area. Créez donc un Area3D que vous appellerez `Fin` et ajoutez un CollisionShape3D.

Figure 24.2 : *Ajout d'un Area de fin*

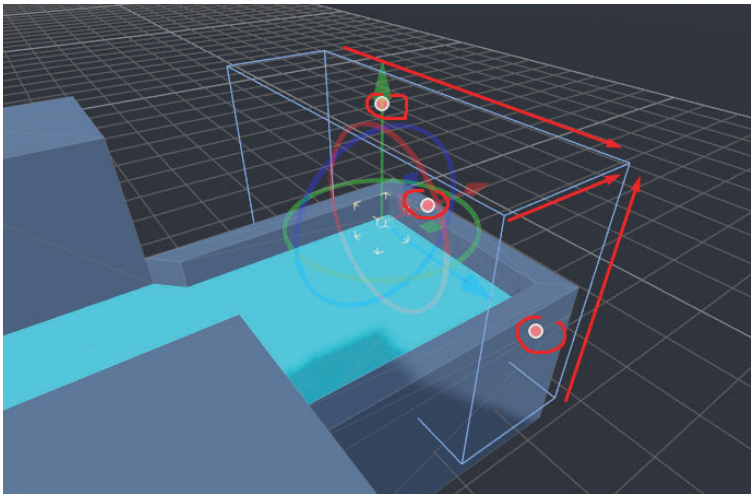
Après avoir ajouté un Area, nous devons spécifier un Shape pour gérer sa collision. Sélectionnez un Shape de type BoxShape3D :

Figure 24.3 : *Ajout du BoxShape3D*



Positionnez ensuite votre Area à l'emplacement de la fin du niveau puis utilisez les petites poignées du CollisionShape pour agrandir la Box de façon à bien couvrir la surface.

Figure 24.4 : *Configuration du BoxShape3D*



Une fois que tout est bien positionné, vous pouvez gérer les collisions. Pensez à cliquer sur l'Area de cet objet afin de le placer dans le bon calque c'est-à-dire `Fin` et à le faire interagir avec le masque `Balle` :

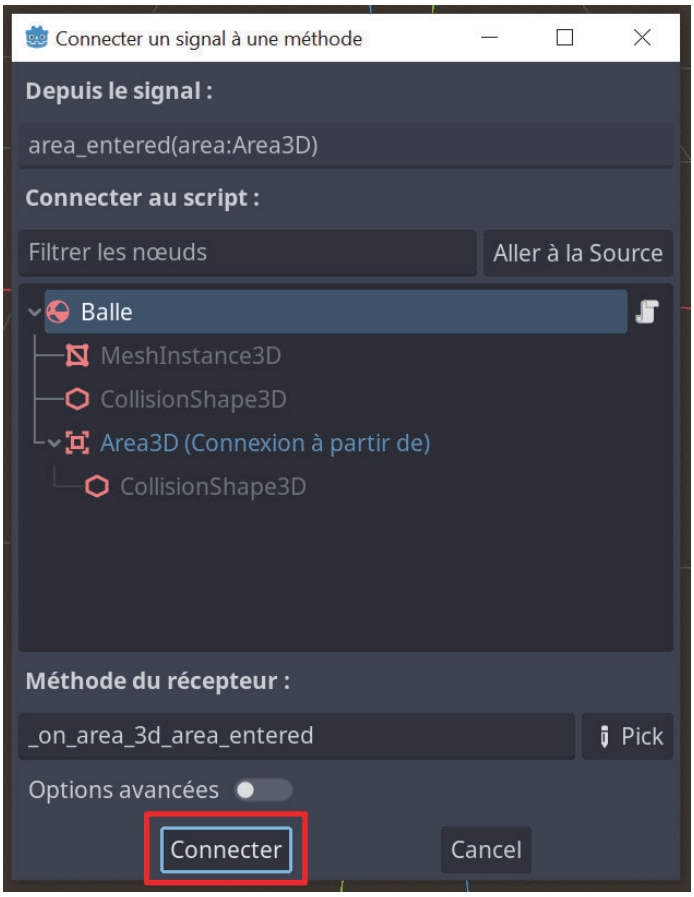
Figure 24.5 : *Configuration des collisions*

Attention › *Si vos calques ne sont pas dans le même ordre que moi, vous ne devrez pas cocher les mêmes cases. Cliquez bien sur la case correspondant au bon calque par rapport à votre projet.*

Pensez aussi à retourner sur la scène de la balle pour faire interagir l'Area de la balle avec les masques `Fin` et `Vide`. Enregistrez et revenez sur le jeu.

Nous allons maintenant écrire la fonction qui se déclenchera lors de la collision entre la balle et l'objet invisible FIN. Cliquez sur la balle puis dépliez les enfants pour cliquer sur le composant Area de la balle. Regardez ensuite dans l'onglet NŒUDS pour trouver l'événement `area_entered` et connectez cet événement au script de la balle.

Figure 24.6 : *Connecter un signal*



Une fois que vous avez connecté ce signal sur la balle, une nouvelle fonction apparaît dans votre script :

```
private void _on_area_3d_area_entered(Area3D area)
{
  // Replace with function body.
}
```

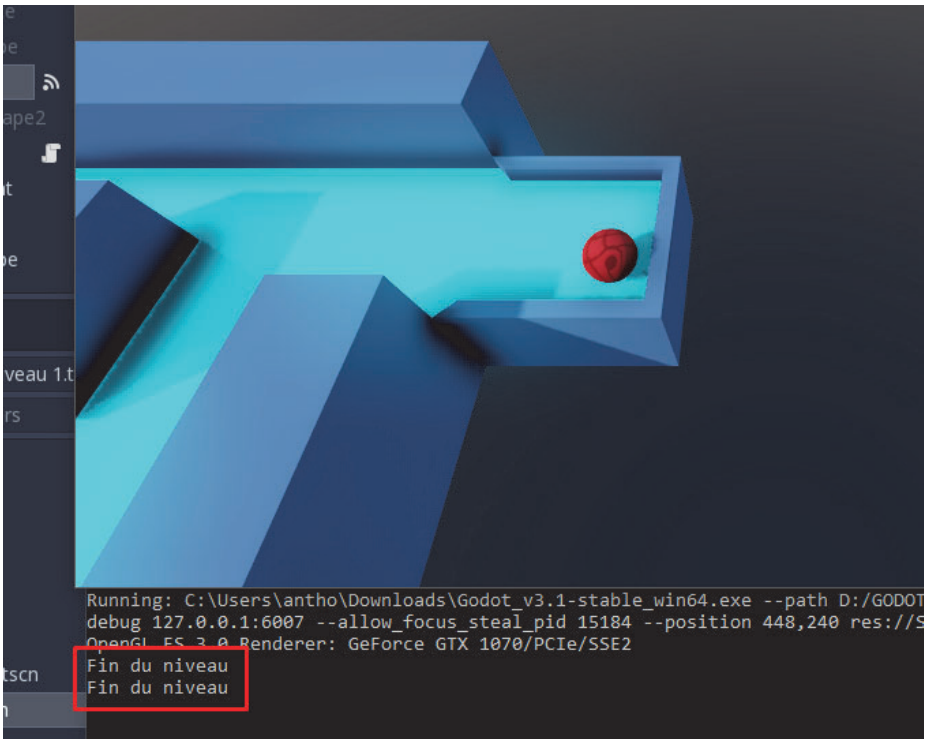
Note *› Godot génère en général les fonctions hors de la classe. Pensez à déplacer la fonction générée à l'intérieur de la classe pour que cela fonctionne.*

Nous allons remplacer le contenu de cette fonction pour vérifier que le nom de l'objet touché est bien `Fin` qui est le nom de l'objet Area que nous avons créé précédemment. Si c'est bien ce nom, nous afficherons un message dans la console :

```
private void _on_area_3d_area_entered(Area3D area)
{
  if (area.Name == "Fin")
  {
    GD.Print("Fin du niveau");
  }
}
```

Testez ce programme :

Figure 24.7 : Test du programme

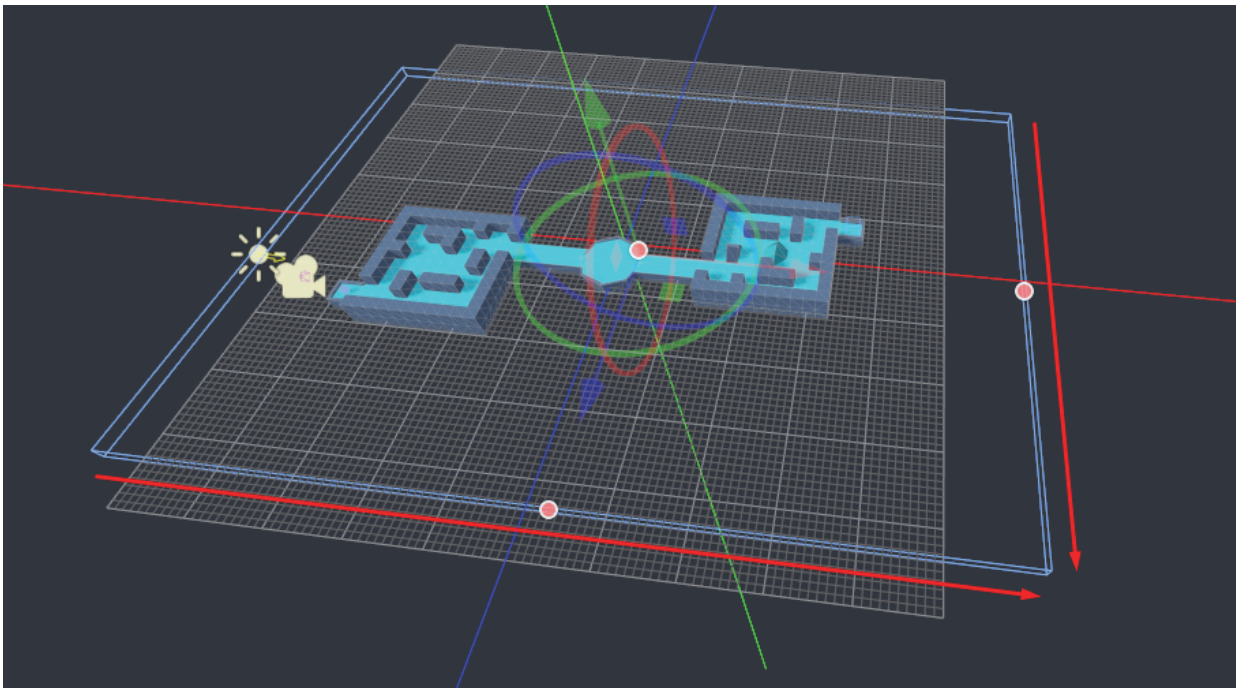


Si vous allez à la fin du niveau et que vous regardez dans la console de débogage, vous verrez le message *Fin du niveau* apparaître. Cela signifie que notre déclencheur fonctionne bien. Nous allons laisser cela tel quel mais, par la suite, vous pourrez afficher un message à l'écran puis lancer le niveau suivant.

24.2. Gérer la chute dans le vide

Passons à la gestion du cas où le joueur tombe dans le vide. Nous allons appliquer exactement la même technique que précédemment. Créez un Area3D puis un CollisionShape3D et définissez BOX pour le SHAPE. Placez cet objet de collision en dessous du niveau et agrandissez la boîte afin qu'elle recouvre toute la surface du niveau comme sur la Figure 24.8.

Figure 24.8 : *Boîte de collision géante*



Vous devez placer cette boîte géante en dessous du niveau afin que la collision ne puisse se produire que si le joueur tombe dans le vide. Prenez de la marge car avec la vitesse le joueur pourrait sauter loin du niveau et vous devez être sûr que quoi qu'il arrive, il touchera forcément cette boîte de collision.

Pensez à renommer l'Area en `Vide` et à bien configurer les calques et masques pour que les collisions soient prises en compte.

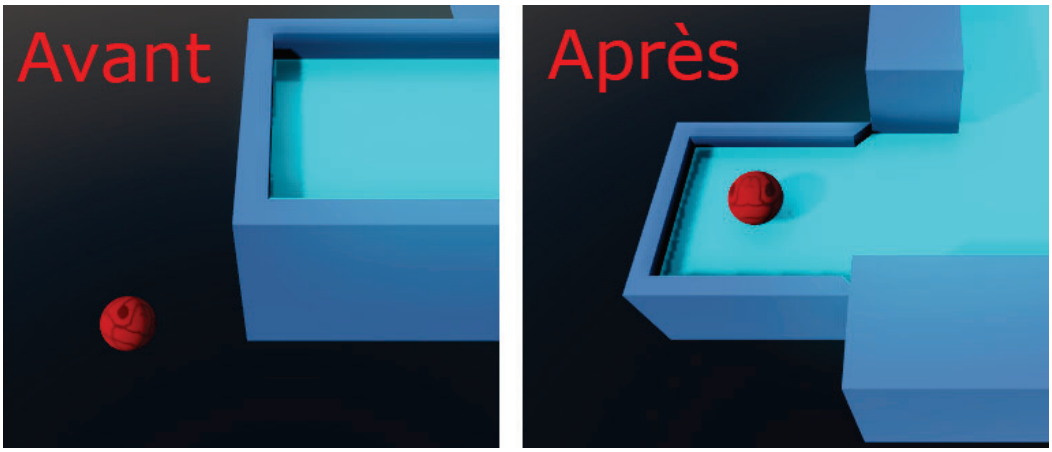
Modifiez ensuite le script de la balle pour tester si la balle tombe dans le vide et rechargez la scène avec la fonction `ReloadCurrentScene` :

```
private void _on_area_3d_area_entered(Area3D area)
{
  // Fin du niveau
  if (area.Name == "Fin")
```

```
    {
      GD.Print("Fin du niveau");
    }
    // Vide
    if (area.Name == "Vide")
    {
      GD.Print("Tombé !");
      GetTree().ReloadCurrentScene();
    }
  }
```

Testez ensuite votre programme pour vérifier le bon fonctionnement de celui-ci :

Figure 24.9 : *Test du programme*



Sur cet exemple, on tombe dans le vide puis le niveau se relance. Voilà comment mettre en place des déclencheurs basiques. Vous pouvez utiliser cette technique pour détecter d'autres collisions avec d'autres éléments (eau, lave, portail magique, téléporteur, etc.).

25

Quelques objets à ramasser

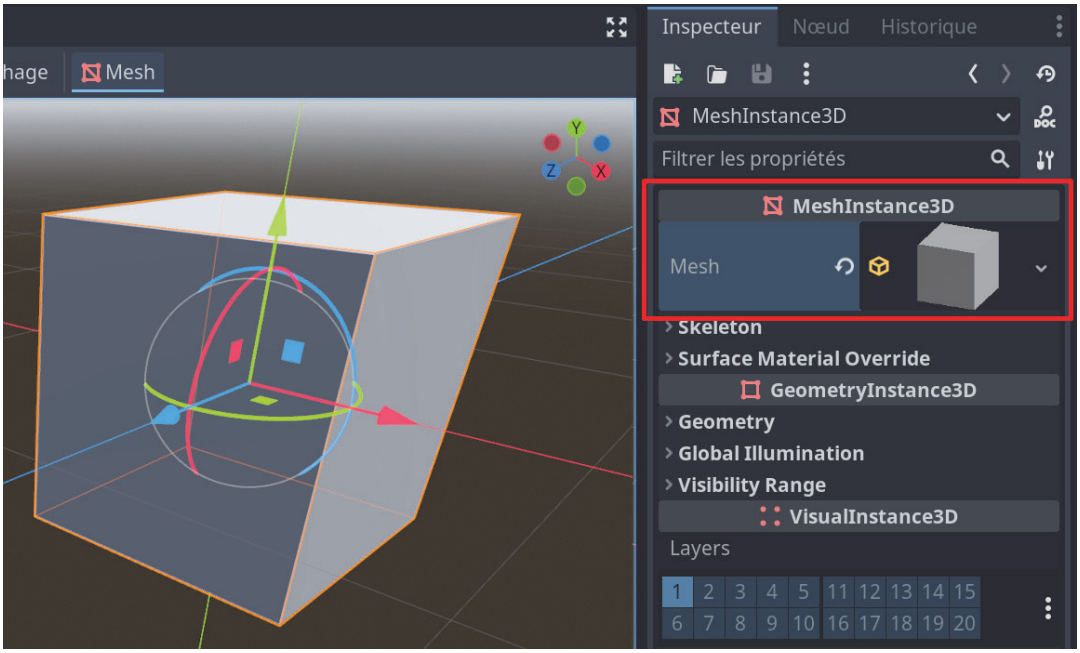
Nous avons désormais les déclencheurs permettant de savoir si le joueur perd où s'il gagne. Nous avons bien avancé mais pour le moment, nous n'avons pas d'objectif mis à part terminer le niveau. L'idéal, pour mesurer la performance du joueur, serait de mettre en place des objets à ramasser afin qu'il y ait un petit challenge. Le joueur pourra soit terminer le niveau d'une traite, soit ramasser le maximum d'objets pour gagner plus de points et ainsi se lancer des défis.

25.1. Création d'un objet à ramasser

Nous allons commencer par créer un objet à ramasser. Il s'agira d'un simple cube. Pour créer celui-ci, créez une nouvelle scène puis créez un cube. Ce cube sera un RigidBody. Ajoutez donc un premier nœud de type RigidBody3D. Pour ne plus avoir l'erreur, nous allons aussi ajouter de suite le composant Mesh et le composant de collision.

Commencez par créer un MeshInstance3D. Via l'inspecteur, utilisez la propriété MESH pour créer un nouveau BoxMesh et ajouter un visuel à votre cube.

Figure 25.1 : Création d'un cube



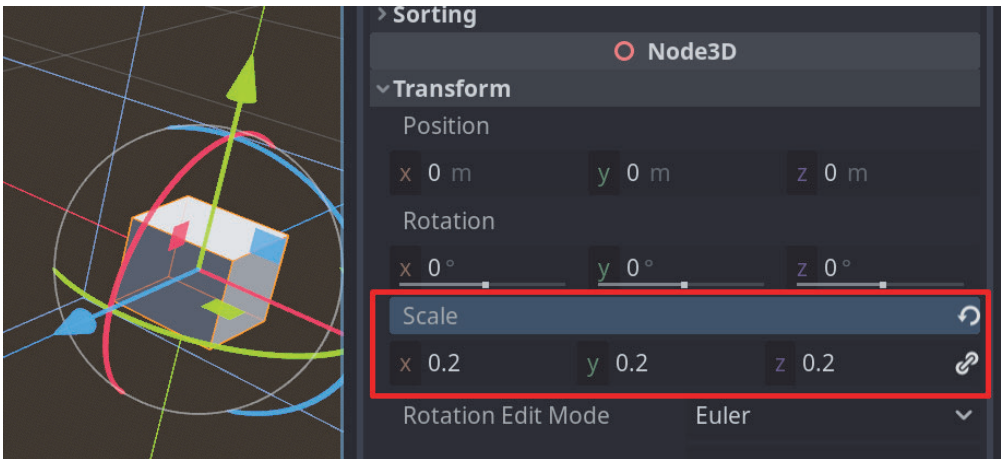
Une fois que vous avez créé le cube, cliquez sur le bouton MAILLAGE en haut de l'écran puis sur CRÉER UNE COLLISION SOEUR CONVEXE SIMPLIFIÉE afin de créer l'objet de collision.

Figure 25.2 : *Les composants du cube*

Note › *Vous pouvez également, si vous le préférez, créer manuellement le COLLISIONSHAPE3D.*

Le cube me semble un peu grand ; en effet, sa taille doit être adaptée à la balle. Cliquez sur le composant MeshInstance et diminuez la taille via l'inspecteur. Vous pouvez ajuster la taille entre 0.2 et 0.5 selon les proportions de vos objets.

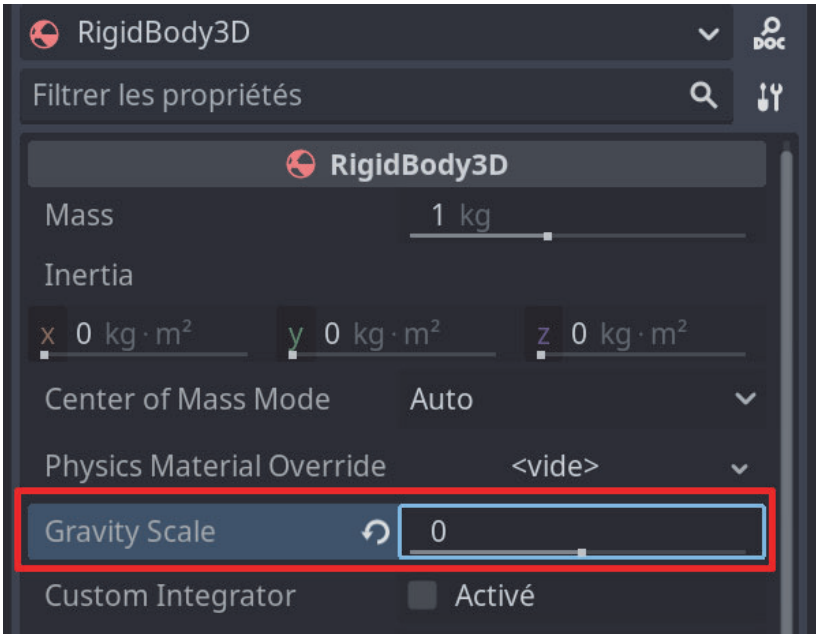
Figure 25.3 : *Modification Scale*



Faites exactement la même chose pour le CollisionShape afin que l'objet de collision soit de la taille du cube. Vous pouvez même le créer légèrement plus grand (0.6 par exemple si la taille du cube est 0.5).

Au niveau de la configuration, nous allons retourner sur le RigidBody. Notre cube doit flotter dans l'air et surtout pas tomber au sol. Par défaut, la gravité s'applique au cube. Pour la désactiver, nous passons à 0 la valeur de la propriété GRAVITY SCALE. Cette manipulation annule la gravité, ainsi le cube flottera.

Figure 25.4 : *Désactivation de la gravité*



Il nous faut ensuite mettre en place les collisions sur ce cube. Nous devons faire en sorte que les collisions entre le cube et la balle soient prises en compte. Ajoutez un calque `Cube` dans les propriétés du projet, puis définissez-le dans les propriétés de collision du RigidBody. Pensez aussi à configurer le masque pour indiquer que le cube interagit avec la balle.

Figure 25.5 : *Préparation des calques*

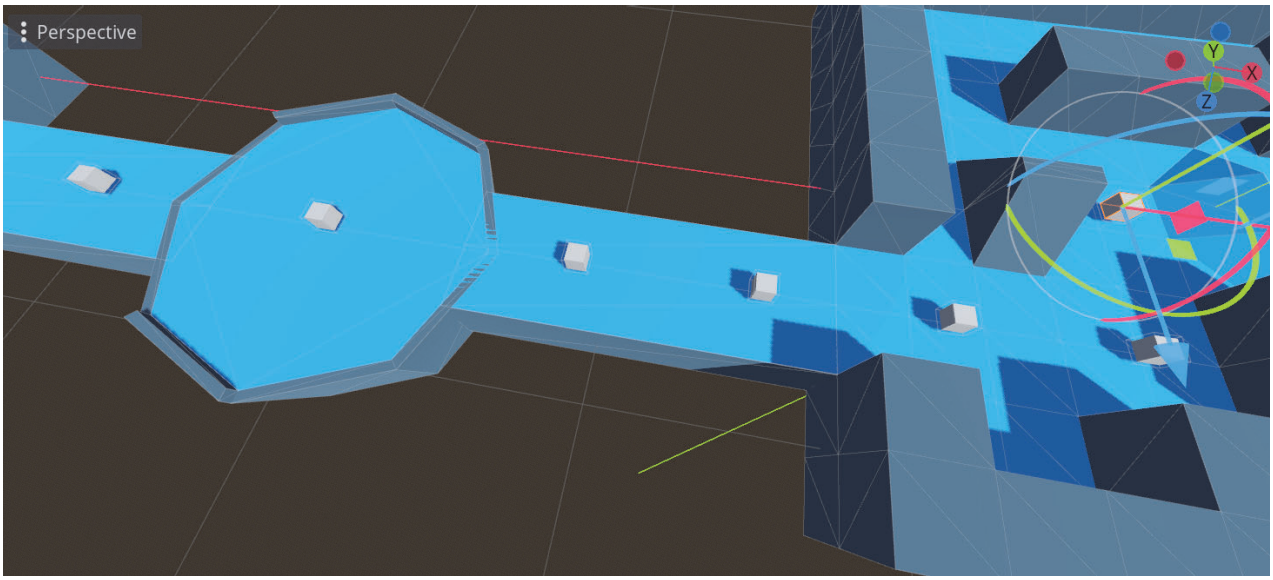
Afin de rendre le jeu un peu plus vivant, nous devons ajouter un minimum de mouvement. C'est pourquoi, pour terminer notre objet, nous allons créer une animation de rotation par script.

Cliquez sur le RigidBody, renommez-le en `Cube`, puis créez un script que vous enregistrerez dans le dossier des scripts. Ce script sera très simple. Vous aurez besoin d'une variable `speed` qui stockera la vitesse et dans la fonction `_Process` vous utiliserez `RotateY` pour faire tourner le cube sur l'axe Y.

```
using Godot;
using System;

public partial class Cube : RigidBody3D
{
  [Export] float speed = 1.0f;

  public override void _Process(double delta)
  {
    RotateY((float)delta * speed);
  }
}
```

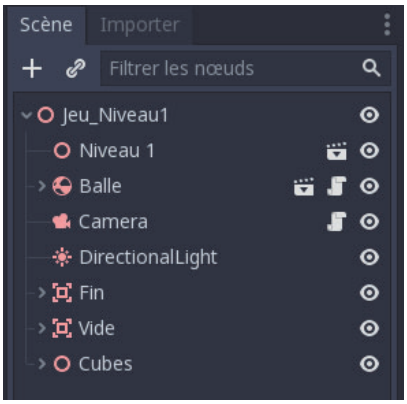
Enregistrez ce script et cette scène puis retournez sur le niveau. Dans le niveau, ajoutez un cube avec l'icône d'instanciation puis dupliquez-le avec Ctrl+D pour en éparpiller plusieurs sur le niveau.

Figure 25.6 : *Ajout de cubes à ramasser*



Organisez aussi la scène en plaçant ces cubes dans un nœud de type Spatial afin de tout regrouper.

Figure 25.7 : Organisation de la scène



Vous pouvez lancer la scène pour tester et vous remarquerez que les cubes sont en mouvement. Pour le moment, impossible de les ramasser !

Note › *Si la vitesse de rotation ne vous convient pas, vous pouvez l'ajuster via l'inspector.*

25.2. Script de collision

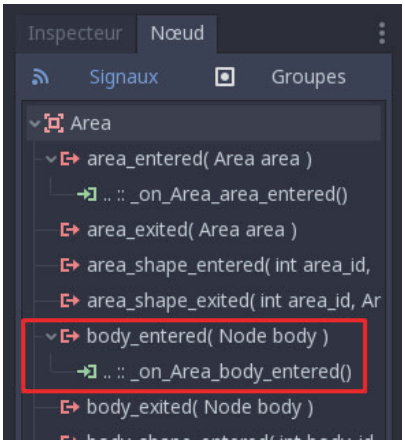
Nous allons maintenant modifier le script de la balle afin de gérer les collisions de celle-ci avec les cubes pour qu'elle puisse les ramasser. Avant cela, pensez à modifier la balle via son Area et configurez les masques pour ajouter le cube. Si vous oubliez d'indiquer que la balle peut interagir avec le cube, alors la collision ne sera pas détectée dans notre script.

Figure 25.8 : Configuration de la balle

Attention › *Il ne suffit pas de cocher les mêmes numéros de calques que sur la Figure 25.8, encore faut-il que vous ayez déclaré dans les paramètres du projet les mêmes calques que moi, et dans le même ordre. Pensez à toujours vérifier le nom du calque correspondant en survolant les cases numérotées avec votre pointeur.*

Enregistrez et retournez sur le niveau. Cliquez sur l'Area de la balle afin de connecter le signal BODY_ENTERED.

Figure 25.9 : *Connexion du signal*



Nous utilisons ce signal car notre cube est un RigidBody et donc c'est ce signal BODY_ENTERED qui se déclenchera. Une fonction est ajoutée à votre script :

```
private void _on_area_3d_body_entered(Node3D body)
{
  // Replacer par le contenu de la fonction
}
```

Dans cette fonction, nous allons détruire le cube ramassé car, une fois ramassé, il doit disparaître et augmenter le nombre de cubes collectés.

Pour détruire un objet, nous utilisons la fonction `QueueFree` qui doit être appliquée sur un objet. L'objet sera `body` en paramètre de la fonction. Cela nous donne :

```
private void _on_area_3d_body_entered(Node3D body)
{
  if(body.Name != "Balle")
  {
    body.QueueFree();
  }
}
```

Note *› Comme la balle est elle-même un RigidBody, j'ai ajouté une condition permettant de l'ignorer car notre Area est en contact avec la balle. Sans cette condition, une collision entre l'Area et un body serait détectée automatiquement.*

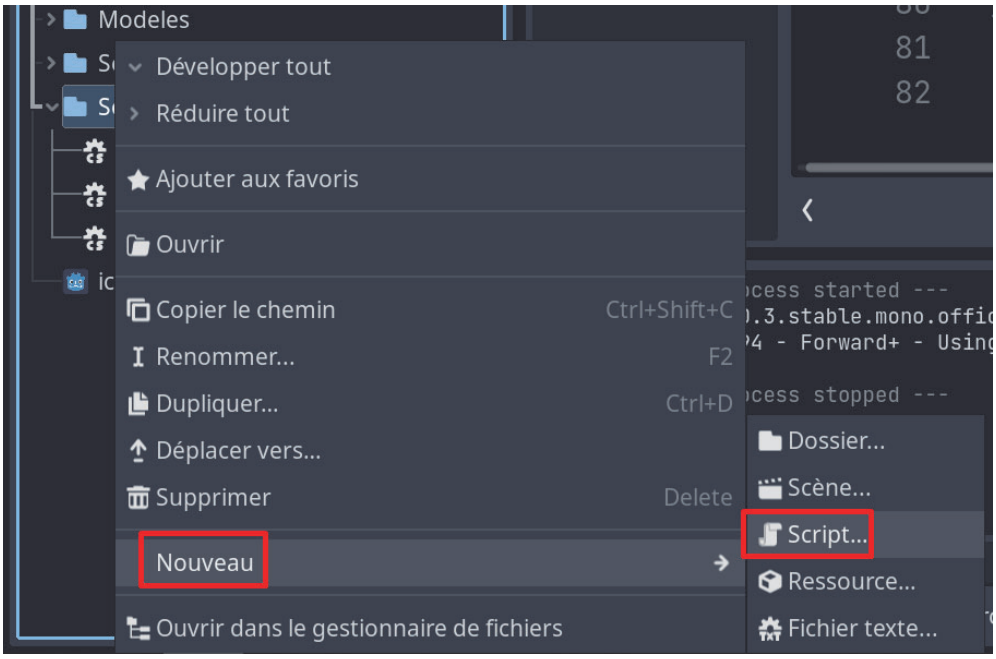
Vous pouvez déjà enregistrer et tester votre jeu. Si tout va bien, une fois touchés, les cubes disparaissent. Si c'est le cas, tout va bien, vous avez correctement configuré votre projet et vous pouvez continuer.

Note *› J'ai volontairement créé les cubes de façon différente par rapport aux autres objets de collision. Cela me permet de vous présenter un autre signal. Mais rien ne vous empêche de procéder comme avec les autres objets (Area3D avec un Mesh et un objet de collision). Il existe toujours plusieurs façons de faire la même chose.*

Nous allons maintenant créer un script qui stockera des données comme le nombre de cubes ramassés. Nous pourrions stocker le nombre de cubes directement dans le script du joueur mais il est plus propre de disposer d'un script indépendant pour l'inventaire. De plus, cela me permettra d'introduire le concept de script global accessible de n'importe où par n'importe quel autre script.

Faites un clic droit sur le dossier des scripts et créez un nouveau script.

Figure 25.10 : *Création d'un script*



Appelez ce script `Global`. Il sera très simple. Pour le moment, il ne servira qu'à stocker le nombre de cubes possédés par le joueur. Créez donc une variable publique `count` pour mémoriser ce nombre :

```
public int count = 0;
```

Puis créez une fonction permettant d'afficher la valeur de cette variable dans la console :

```
public void PrintCount()
{
  GD.Print(count);
}
```

Enfin, créez une fonction qui permet d'incrémenter cette variable :

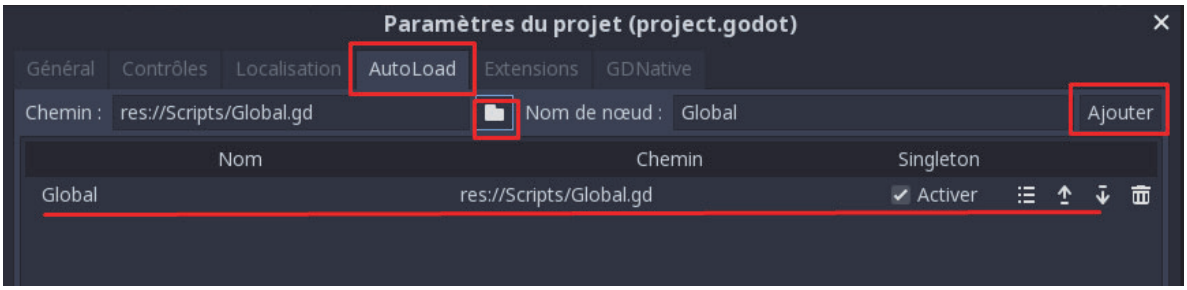
```
public void IncreaseCount()
{
  count = count + 1;
}
```

Ce qui nous donne le code suivant :

```
using Godot;
using System;

public partial class Global : Node
{
  public int count = 0;

  public void PrintCount()
  {
    GD.Print(count);
  }

  public void IncreaseCount(){
    count = count + 1;
  }
}
```

Ce script est terminé. Enregistrez-le. Nous ne le mettrons sur aucun objet mais nous allons le charger automatiquement au lancement du jeu. Pour charger un script de cette façon, allez dans les paramètres du projet puis rendez-vous dans l'onglet AUTOLOAD. Ici dans le chemin, chargez le script `Global` puis cliquez sur AJOUTER pour l'ajouter à la liste des scripts automatiquement chargés.

Figure 25.11 : *Chargement du script*

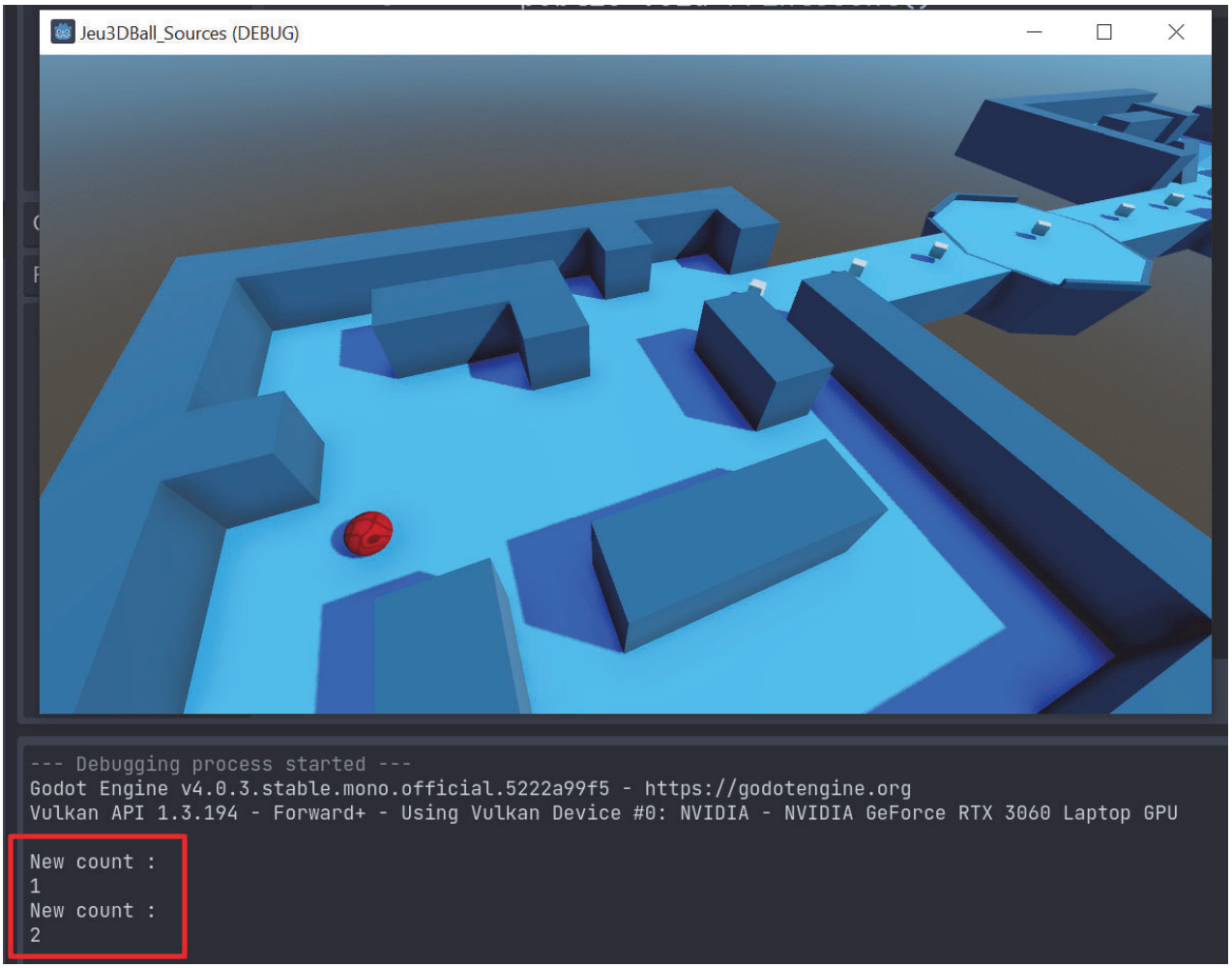


Maintenant, il est possible d'accéder à ce script et à ses fonctions à partir de n'importe quel script. Nous allons utiliser cela dans le script de la balle pour incrémenter le nombre de cubes et afficher le nombre de cubes possédés. Voici le code de la fonction de collision :

```
private void _on_area_3d_body_entered(Node3D body)
{
  if(body.Name != "Balle")
  {
    body.QueueFree();
    // Modification de la variable count du script Global
    var globalScript = GetNode<Global>("/root/Global");
    globalScript.IncreaseCount();
    GD.Print("New count : ");
    globalScript.PrintCount();
  }
}
```

Dans l'ordre, nous détruisons le cube, nous créons une référence vers le script `Global` (cette référence peut être une variable en haut du script pour que ce soit plus propre), nous incrémentons le nombre de cubes puis nous affichons le nombre de cubes ramassés dans la console. Voici ce que cela donne en test :

Figure 25.12 : Affichage dans la console



Comme vous pouvez le constater, notre script fonctionne et le nombre de cubes ramassés s'affiche bien dans la console. Notre script global remplit bien sa fonction. Grâce à ce système vous aurez accès de n'importe où au nombre de cubes. Vous pouvez par exemple vous en servir pour créer une porte qui ne s'ouvre qu'avec dix cubes afin de forcer le joueur à ramasser le maximum de cubes possible.

Pour notre projet, nous allons nous arrêter ici mais vous pouvez utiliser ce système pour créer d'autres objets à ramasser comme des clés ou des bonus permettant par exemple d'aller plus vite.

26

L'interface utilisateur

Dans la quasi-totalité des jeux, nous pouvons retrouver une interface utilisateur. Cette interface permet d'afficher des informations à l'écran comme la vie, l'énergie, la barre de compétences, le menu pause ou d'autres éléments. Dans notre cas, il serait intéressant d'afficher le nombre de cubes ramassés ainsi que le temps passé dans le niveau afin de proposer au joueur un challenge supplémentaire : aller le plus vite possible.

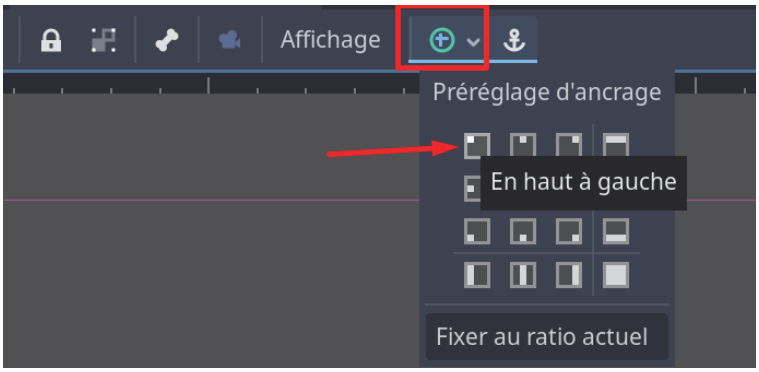
26.1. Affichage du compteur de cubes et du temps

Commençons par l'affichage du nombre de cubes. Créez une nouvelle scène sur laquelle vous allez construire l'interface. Ajoutez un nœud de type INTERFACE UTILISATEUR (Control). Le nœud Control est le nœud de base que nous utilisons pour construire une interface utilisateur. En comparaison, le nœud Spatial est le nœud de base pour construire une scène 3D. Une fois le nœud Control créé, ajoutez un Label.

Figure 26.1 : *Ajout du label*

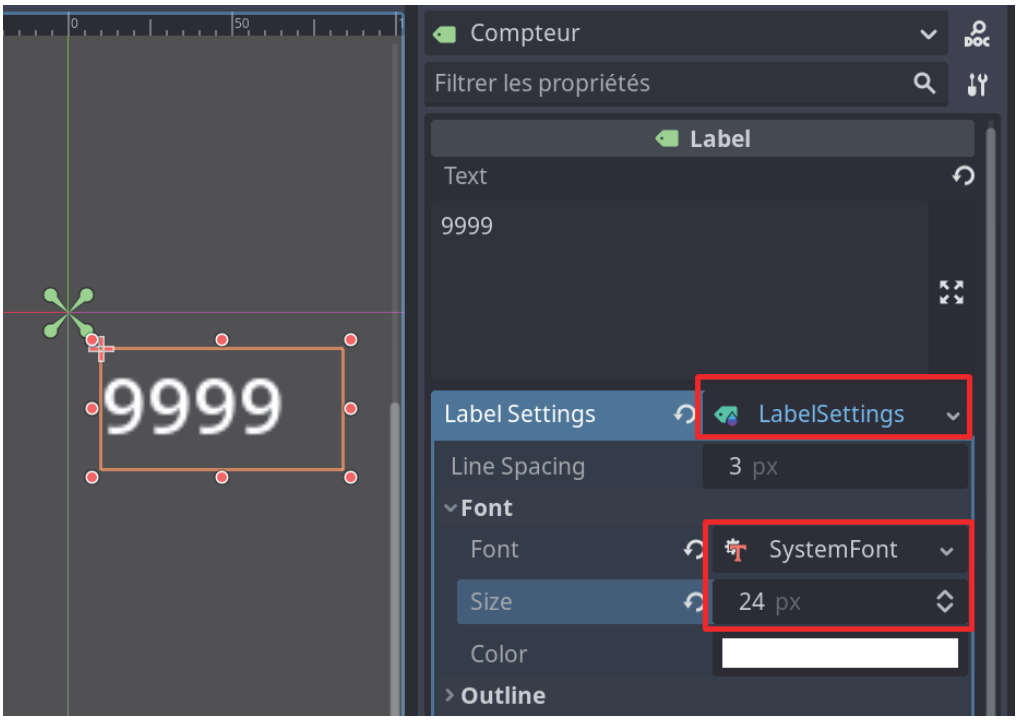
Placez le label en haut à gauche grâce à l'outil d'ancrage :

Figure 26.2 : *Ancrage du label*



Renommez le nœud principal en `GUI` et le label en `Compteur`. Vous pouvez ajouter un texte fictif comme `9999` afin d'avoir un visuel pour travailler. Maintenant, retournez sur le label et ajustez la police d'écriture. Créez un `LabelSettings` suivi d'un `SystemFont` afin d'avoir accès à la propriété `SIZE` pour ajuster la taille.

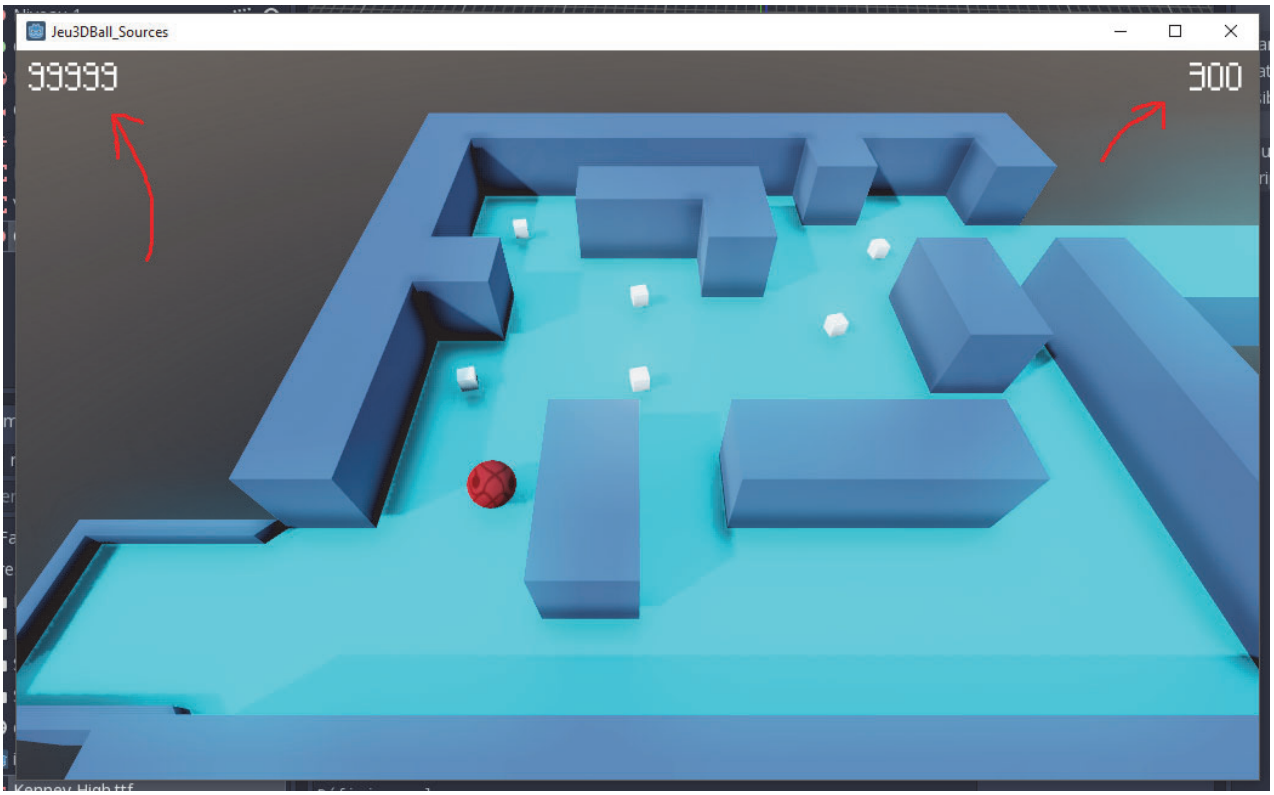
Figure 26.3 : *Police personnalisée*



La seule chose que j'ai faite, c'est agrandir la taille de la police. Cela nous permettra de mieux voir le texte. Si vous souhaitez aller plus loin et personnaliser encore plus votre texte, faites-le.

Ceci est très basique mais sera largement suffisant pour notre projet. Dupliquez ce label et placez la copie en haut à droite. Renommez cette copie en `Temps`. C'est sur ce second label que nous afficherons les secondes écoulées. Une fois fait, enregistrez votre scène `GUI` et instanciez-la dans le niveau 1.

Figure 26.4 : Ajout du GUI dans le jeu



Maintenant que l'interface est ajoutée au jeu, il ne nous reste plus qu'à programmer la mise à jour de cette interface.

26.2. Programmation du compteur

Commençons par afficher le nombre de cubes ramassés. Dans le niveau 1, ouvrez le script de la balle. Nous allons ajouter une variable qui stockera le label associé au compteur. La valeur de cette variable sera initialisée au lancement du jeu (dans la fonction `_Ready`). Pour récupérer un nœud dans la scène, nous utilisons `GetParent().GetNode()`. Cela permettra de rechercher un nœud. Il faut ensuite passer en paramètre le nom de ce nœud de la façon suivante :

```
Label cpt;

public override void _Ready()
{
  cpt = GetParent().GetNode("GUI/Compteur") as Label;
}
```

Attention › *Pour que la recherche fonctionne, veillez à bien respecter l'arborescence des nœuds. Si vos nœuds sont nommés ou placés différemment par rapport aux miens (voir Figure 26.5), alors ajustez le chemin vers le nœud recherché.*

Figure 26.5 : Mon organisation

Nous avons maintenant un raccourci vers le label. Pour modifier le texte de celui-ci, utilisez simplement `cpt` (le nom de la variable que nous venons de créer) et utilisez sa propriété `Text`. Par exemple, dans la fonction `_Ready` que nous pouvons créer, nous allons initialiser la variable à zéro :

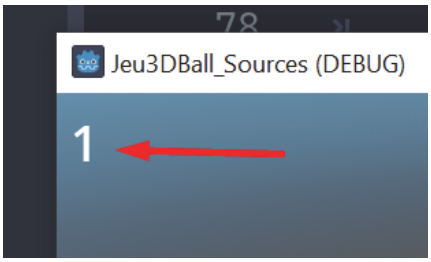
```
public override void _Ready()
{
  cpt = GetParent().GetNode("GUI/Compteur") as Label;
  cpt.Text = "0"; // Initialisation à 0
}
```

Nous pouvons faire de même dans la fonction de collision avec les cubes :

```
private void _on_area_3d_body_entered(Node3D body)
{
  if(body.Name != "Balle")
  {
    body.QueueFree();
    // Modification de la variable count du script Global
    var globalScript = GetNode<Global>("/root/Global");
    globalScript.IncreaseCount();
    GD.Print("New count : ");
    globalScript.PrintCount();
    // Affichage sur le GUI
    cpt.Text = globalScript.count.ToString();
  }
}
```

Ici, nous devons utiliser la fonction `ToString` pour convertir le nombre en texte car le label n'est compatible qu'avec du texte. Vous pouvez alors tester votre programme. Après avoir ramassé un cube, le compteur s'incrémente comme prévu.

Figure 26.6 : *Ramassage d'un cube*

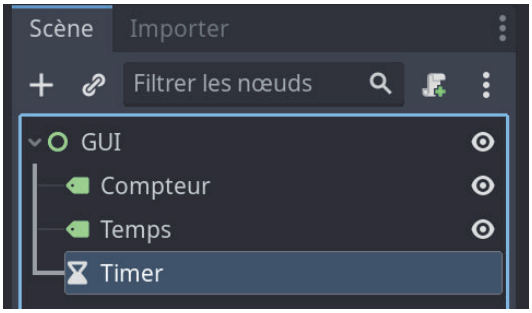


26.3. Programmation du chronomètre

Les joueurs aiment se lancer des défis. Parfois, même sans un objectif clair, les joueurs essayent de se surpasser. Par exemple Minecraft est amusant par pour ses objectifs mais pour ses possibilités infinies. Il est possible de construire ce que l'on souhaite. Les joueurs essayent donc de créer les structures les plus impressionnantes possibles. Dans notre cas, le défi serait par exemple de finir le niveau le plus rapidement possible sans tomber. Pour cela, nous devons calculer le temps écoulé.

Passons donc à la création du chronomètre. Nous allons retourner sur la scène `GUI` et y ajouter un nœud de type Timer. Ce composant permet de déclencher un événement à un certain intervalle de temps. Dans notre cas, chaque seconde, nous allons ajouter `1` au temps écoulé.

Figure 26.7 : *Ajout d'un Timer*



Ajoutez un script au nœud `GUI`. Dans ce script, nous ajoutons une variable permettant de stocker le temps :

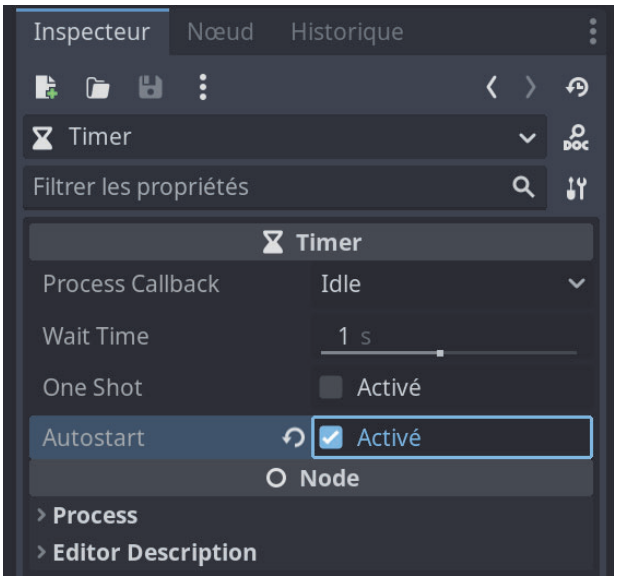
```
int time = 0;
```

Créez ensuite une fonction d'incrémentation du temps. Celle-ci incrémente le temps et met à jour l'interface. Pensez à mettre `0` comme texte par défaut.

```
public void IncrementTime()
{
  time = time + 1;
  Label label = GetNode("Temps") as Label;
  label.Text = time.ToString();
}
```

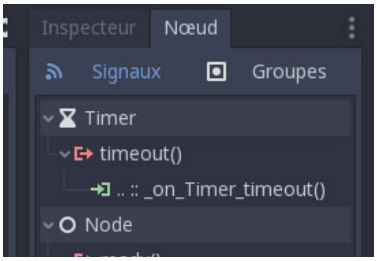
Cliquez ensuite sur le Timer et, dans l'inspecteur, cochez la propriété AUTOSTART afin qu'il se déclenche automatiquement. Parmi les autres propriétés, vous trouverez WAIT TIME qui correspond au temps d'attente avant la fin du timer. Ici, nous restons sur 1 seconde mais vous pouvez modifier cela pour créer un timer avec un temps différent.

Figure 26.8 : Autostart



Dans le panneau NŒUDS, connectez l'événement `timeout()` au nœud `GUI`.

Figure 26.9 : Signal Timeout



Note › *L'événement Timeout se déclenche dès que le temps du timer est écoulé. Dans notre cas, l'événement est déclenché au bout de 1 seconde. Nous répétons cela à l'infini afin de compter le temps, seconde après seconde.*

Une nouvelle fonction est créée dans le script. Appelez alors votre fonction d'incrémentation :

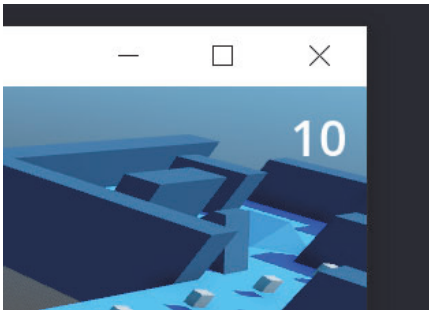
```
private void _on_timer_timeout()
{
  IncrementTime();
}
```

Ce qui nous donne le script complet suivant :

```
using Godot;
using System;

public partial class gui : Control
{
  int time = 0;

  public void IncrementTime()
  {
    time = time + 1;
    Label label = GetNode("Temps") as Label;
    label.Text = time.ToString();
  }

  private void _on_timer_timeout()
  {
    IncrementTime();
  }
}
```

Enregistrez tout cela et retournez dans votre jeu. Lancez-le avec la touche F6 afin de tester que tout fonctionne comme prévu. Le temps s'écoule tel que souhaité !

Figure 26.10 : *Notre timer*



Nous avons donc une interface simple qui affiche le nombre de cubes ramassés ainsi que le temps écoulé. Pour ce livre, je vais m'arrêter ici. Le but de ce chapitre était de vous donner quelques clés pour implémenter du challenge dans vos jeux et créer des variables pour mesurer des quantités. Avec cela, vous pourrez mettre en place des obstacles, définir des défis, créer des succès à débloquer (ramasser dix cubes en dix secondes), etc.

Vous avez également les outils pour créer un menu principal au lancement du jeu. Vous pouvez créer une interface avec une image de fond et un bouton JOUER.

Figure 26.11 : *Exemple de menu*

Encore une fois, il s'agit d'un menu basique avec une image plein écran et un bouton JOUER cliquable. Je ne détaille pas car vous savez réaliser ce menu. Vous aurez besoin d'un TextureRect et d'un Label. Vous devez créer un script pour le bouton JOUER et connecter le signal `pressed()`. Le code du bouton sera :

```
private void _on_button_jouer_pressed()
{
  GetTree().ChangeSceneToFile("res://Scenes/Jeu_Niveau1.tscn");
}
```

Vous pourrez également afficher dans le jeu un message *Gagné* ou *Perdu* en fonction de ce que fait le joueur.

Vous voilà avec un certain nombre de pistes pour améliorer votre projet ; continuez de pratiquer de votre côté et créez différents niveaux avec différentes variations.

27

Finalisation et publication du jeu

Dans ce chapitre, nous allons finaliser notre jeu en ajoutant un petit son lorsqu'on ramasse les cubes. Puis nous le compilerons et le diffuserons sur Internet.

27.1. Émettre un son quand un objet est ramassé

Téléchargez un son libre de droits et importez-le dans votre projet Godot. Souvenez-vous que Godot ne prend en compte que les sons de type WAV ou OGG.

Ouvrez le script de la balle et ajoutez-y la variable suivante :

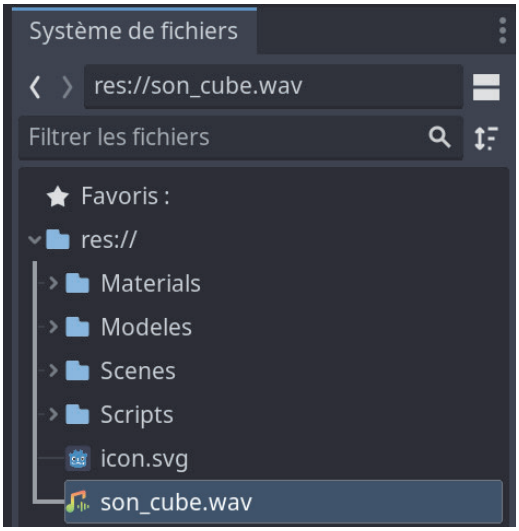
```
AudioStream cubeSound;
```

Vous l'aurez compris, cette variable stockera le son émis par le cube lorsqu'il est ramassé. Dans la fonction `_Ready`, chargez le son.

```
cubeSound = GD.Load("res://son_cube.wav") as AudioStream;
```

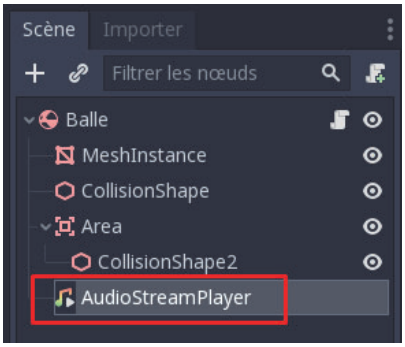
Notre exemple de code implique que le son s'appelle `son_cube.wav` et qu'il est placé à la racine du projet Godot.

Figure 27.1 : Le son



Nous pouvons maintenant jouer le son. Pensez-bien à ajouter un composant AudioStreamPlayer3D à la balle (en passant par la scène de la balle). C'est ce composant qui sera capable de produire un son. Sans lui, pas de son ! Relisez le chapitre Musique et effets sonores si besoin.

Figure 27.2 : *AudioStreamPlayer3D*



Retournez dans le jeu et le script de la balle. Ajoutez une variable pour l'AudioStreamPlayer :

```
AudioStreamPlayer3D audioPlayer;
```

Dans `_Ready` renseignez cette variable :

```
audioPlayer = GetNode<AudioStreamPlayer3D>("AudioStreamPlayer3D");
```

Dans la fonction de collision avec le cube, spécifiez le son qui doit être joué avec le code suivant :

```
audioPlayer.Stream = cubeSound;
```

`audioPlayer` permet d'accéder au composant. `Stream` permet d'accéder à la propriété STREAM qui correspond au son. Après cela, lancez le son avec `Play` :

```
audioPlayer.Play();
```

Puis testez ! Tout devrait fonctionner. Pour rappel, voilà le code complet de la balle qui est le script le plus complexe du projet :

```
using Godot;
using System;

public partial class balle : RigidBody3D
{
  [Export] public float force = 1f;
```

```
Label cpt;
AudioStream cubeSound;
AudioStreamPlayer3D audioPlayer;

public override void _Ready()
{
  cpt = GetParent().GetNode("GUI/Compteur") as Label;
  cpt.Text = "0";
  audioPlayer = GetNode<AudioStreamPlayer3D>("AudioStreamPlayer3D");
  cubeSound = GD.Load("res://son_cube.wav") as AudioStream;
}

public int GetAxis(string axis)
{
  // Si on effectue un mouvement horizontal
  if (axis == "horizontal")
  {
    // Flèche de gauche
    if (Input.IsActionPressed("ui_left"))
      return -1;
    // Droite
    else if (Input.IsActionPressed("ui_right"))
      return 1;
  }
  // Si vertical
  else if (axis == "vertical")
  {
    // Haut
    if (Input.IsActionPressed("ui_up"))
      return -1;
    // Bas
    else if (Input.IsActionPressed("ui_down"))
      return 1;
  }
  return 0;
}

public override void _PhysicsProcess(double delta)
{
  // Mouvement en X
  int xSpeed = GetAxis("horizontal");
  // Mouvement en Y
  int ySpeed = GetAxis("vertical");
  // Vecteur 3 de direction
  var dir = new Vector3(xSpeed, 0, ySpeed).Normalized();
  // Pour la vitesse de mouvement
  float magnitude = force * (float)delta;
  // On applique la force de propulsion dans la direction * la force
  AddImpulse(magnitude * dir);
}

public void AddImpulse(Vector3 f)
{
  // Si une force existe en X ou Y
  if (f[0] != 0 || f[2] != 0)
```

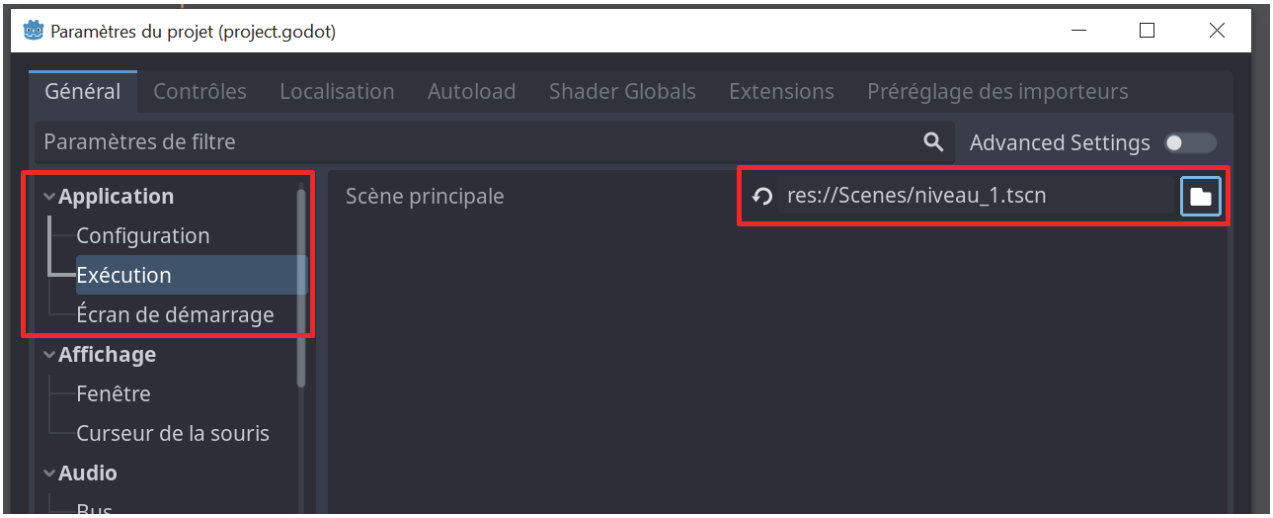
```
        {
            // On ajoute une impulsion à la balle
            ApplyCentralImpulse (f);
        }
    }

    private void _on_area_3d_area_entered(Area3D area)
    {
        // Fin du niveau
        if (area.Name == "Fin")
        {
            GD.Print("Fin du niveau");
        }
        // Vide
        if (area.Name == "Vide")
        {
            GD.Print("Tombé !");
            GetTree().ReloadCurrentScene();
        }
    }

    private void _on_area_3d_body_entered(Node3D body)
    {
        if(body.Name != "Balle")
        {
            // Jouer un son
            audioPlayer.Stream = cubeSound;
            audioPlayer.Play();
            // Destruction du cube
            body.QueueFree();
            // Modification de la variable count du script Global
            var globalScript = GetNode<Global>("/root/Global");
            globalScript.IncreaseCount();
            GD.Print("New count : ");
            globalScript.PrintCount();
            // Affichage sur le GUI
            cpt.Text = globalScript.count.ToString();
        }
    }

}
```

Vous avez maintenant du son lorsque vous touchez les cubes. Nous avons terminé notre jeu ! Bien sûr, vous devez, de votre côté, créer plusieurs niveaux et les lier de façon à lancer le niveau suivant lorsque le joueur termine l'actuel. Pour mon exemple, le jeu est constitué d'un seul niveau. Nous allons maintenant le compiler pour pouvoir le partager en ligne.

27.2. Compilation du jeu

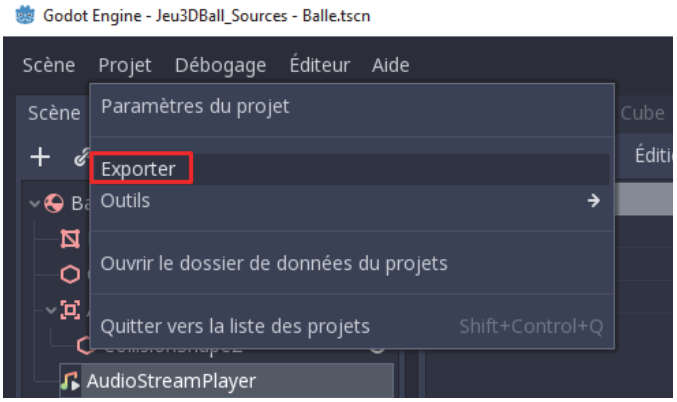
Commencez par vous rendre dans les paramètres du projet sous la rubrique APPLICATION/EXÉCUTION afin de spécifier quelle scène doit se lancer au démarrage du jeu.

Figure 27.3 : *Choix de la scène principale*



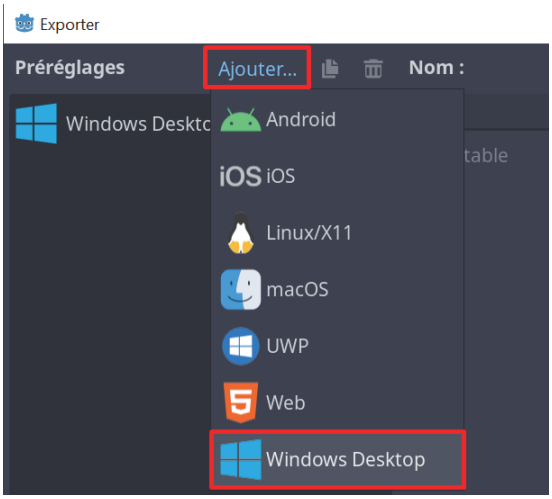
Allez ensuite dans le menu PROJET/EXPORTER.

Figure 27.4 : *Exportation du projet*



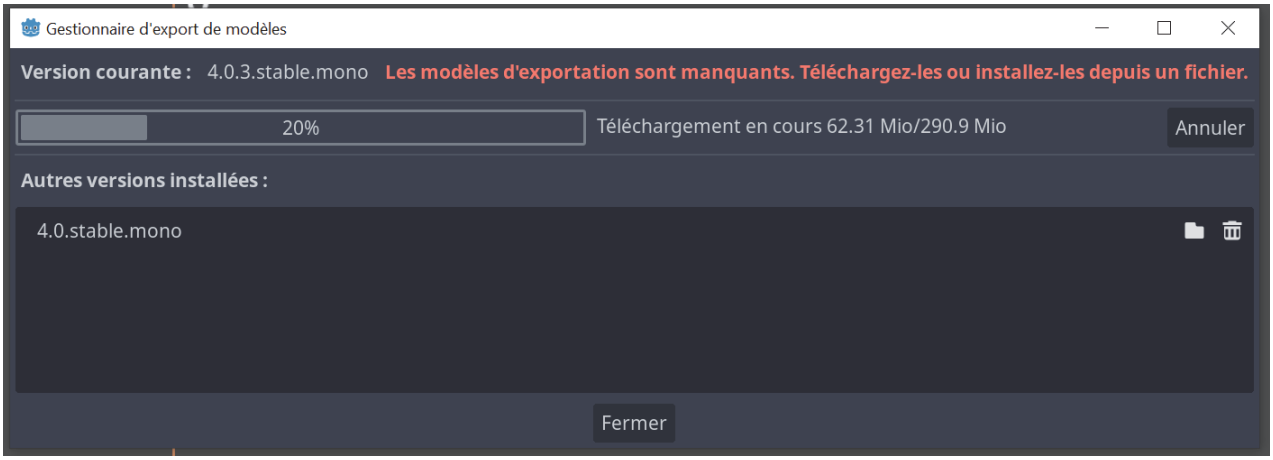
Ici, cliquez sur AJOUTER afin d'ajouter une plateforme d'exportation. Vous pouvez créer un projet HTML5, c'est-à-dire une version navigateur web du jeu, ou WINDOWS DESKTOP pour créer une version Windows. Vous pouvez créer également une version macOS ou Linux. Pour ma part, je choisis Windows.

Figure 27.5 : *Export Windows*



Si aucun modèle d'exportation n'est trouvé, cliquez sur le lien GÉRER LES MODÈLES D'EXPORTATION fourni dans le message d'erreur, et téléchargez-en un. Cela peut prendre plus ou moins de temps car il y a plusieurs centaines de mégas à télécharger.

Figure 27.6 : *Téléchargement du modèle*



Une fois terminé, retournez sur la fenêtre d'exportation et cliquez sur EXPORTER pour générer l'exécutable de votre jeu.

Figure 27.7 : *Exportation de notre jeu*

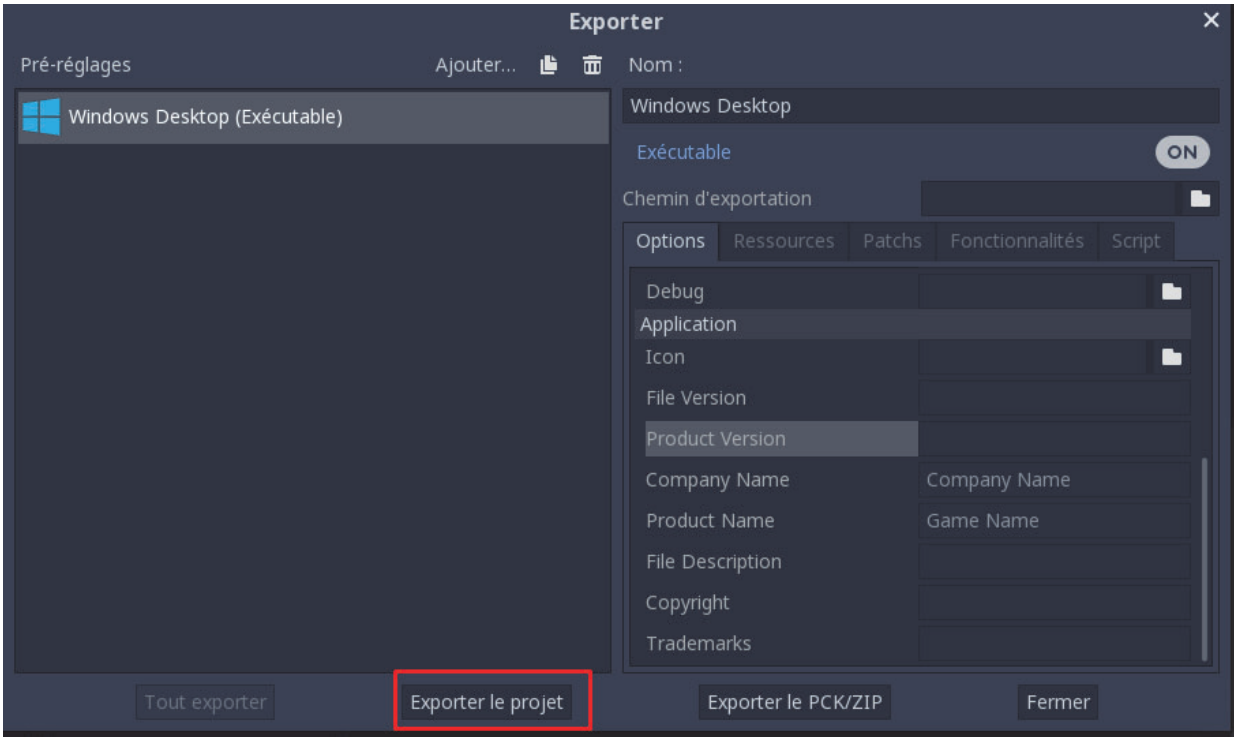
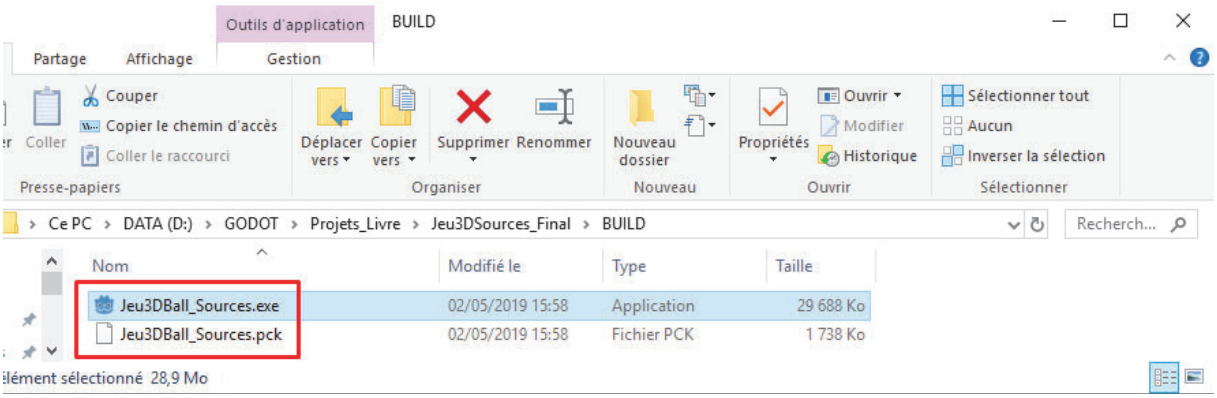
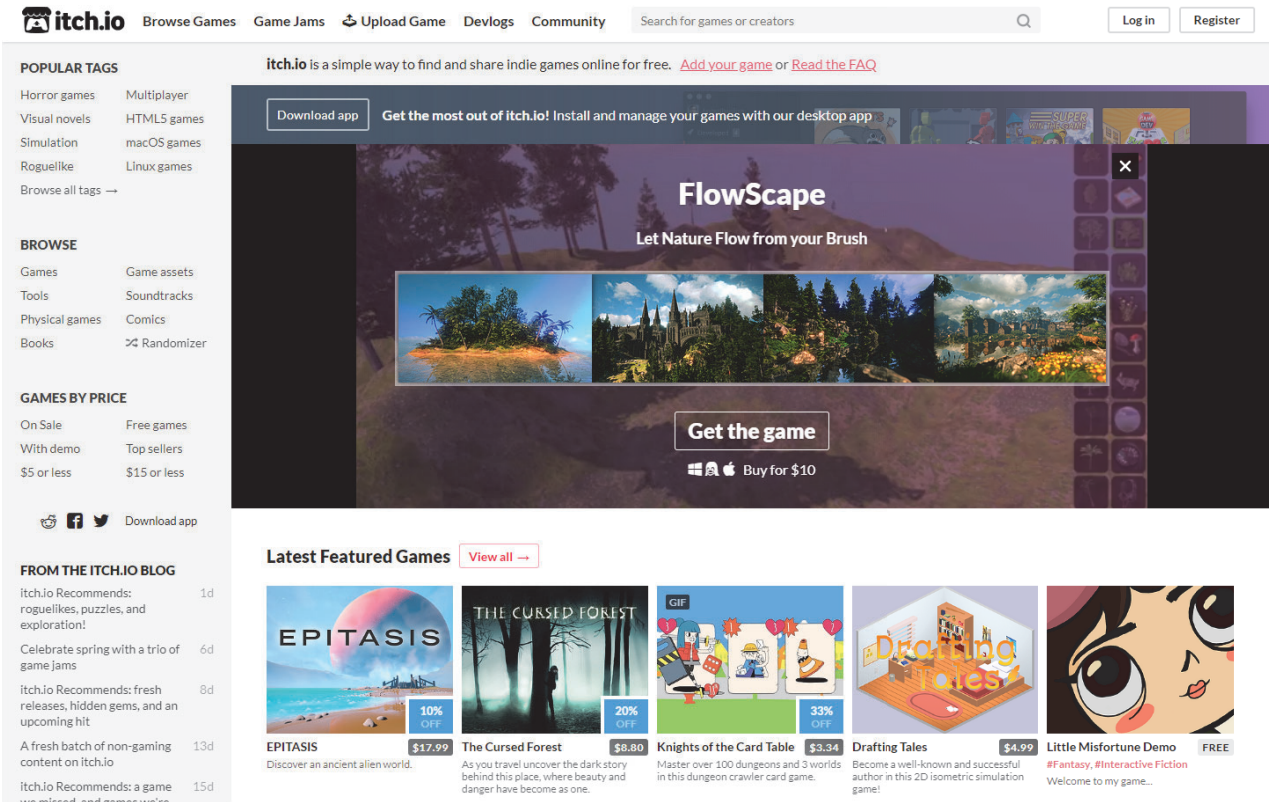


Figure 27.8 : *Exécutables de notre jeu*



Vous avez réussi à compiler votre jeu, vous pouvez désormais le diffuser ! Le plus simple pour le partager est de compresser les fichiers `.exe` et `.pck` dans un fichier ZIP que vous pourrez ensuite distribuer sur des plateformes de jeux indépendants comme par exemple itch.io.

Figure 27.9 : *itch.io*



Itch.io est une plateforme sur laquelle vous pouvez publier gratuitement vos jeux pour les partager ou les vendre. La procédure est très simple. Créez un compte et depuis votre tableau de bord vous aurez accès à la zone permettant de publier un jeu. Il vous faudra définir le titre, la description, créer des captures d'écran, choisir un prix et envoyer le fichier zip de votre jeu. C'est selon moi la plateforme parfaite pour les indépendants qui souhaitent se lancer et avoir des retours.

Vous avez donc découvert comment créer un jeu de A à Z avec Godot, comment générer l'exécutable et comment le publier afin de partager votre création avec de nombreux joueurs . Vous avez maintenant toutes les clés en main pour créer de nombreux jeux qui auront, je l'espère, beaucoup de succès !

Aller plus loin avec Godot

Dans ce livre, vous avez découvert comment créer vos jeux 2D et 3D avec Godot. Avec ces bases, vous pouvez vous lancer sur vos propres projets. Vous pouvez évidemment aller encore plus loin et poursuivre votre apprentissage. Je vais vous donner quelques pistes pour continuer de progresser.

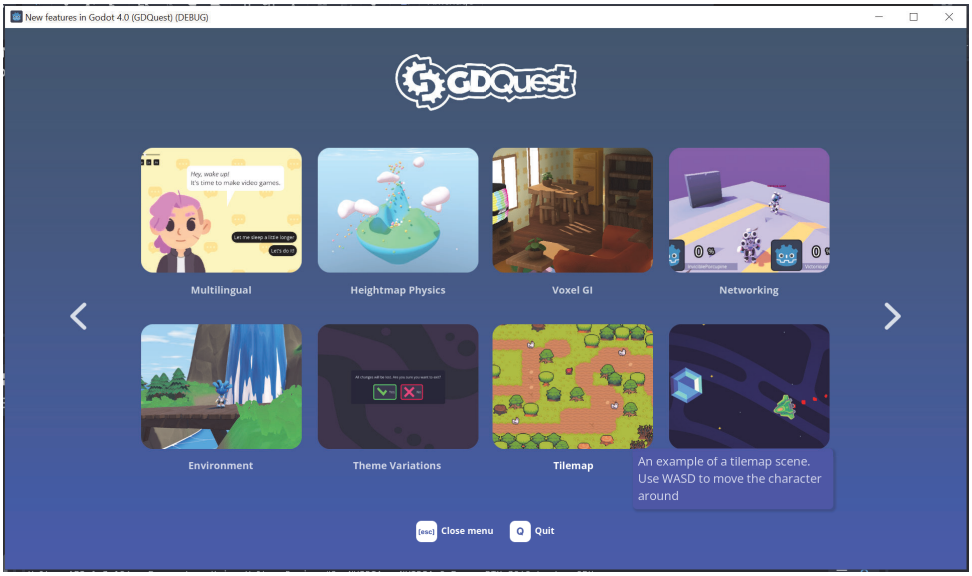
28

Les pistes pour s'améliorer

- Sur internet, vous trouverez de nombreux tutoriels vidéo sur l'utilisation de Godot. Pour peu que vous compreniez l'anglais, vous disposerez d'un vaste choix en complément de ce livre. Cela vous permettra d'apprendre et de pratiquer avec des exemples différents et de découvrir des nœuds complémentaires.
- La documentation officielle de Godot est également une très bonne source de connaissances. Vous y retrouverez bien sûr toute la documentation sur Godot, ses nœuds, ses outils et son utilisation mais aussi des mini tutoriels qui vous permettront de travailler sur des exemples précis.
- Les game jams sont une très bonne opportunité pour travailler sur des petits jeux de façon intensive sur un court laps de temps. C'est très formateur et c'est également l'occasion de rencontrer d'autres passionnés. Il existe des sites comme itch ou GodotWildJam qui proposent régulièrement des jams. C'est une expérience à tenter au moins une fois dans sa vie.
- Vous pouvez apprendre en suivant des tutoriels (vidéo ou écrits) mais aussi en décortiquant des projets existants. Certaines personnes préfèrent regarder comment est construit un projet afin de pouvoir le reproduire de la même façon.

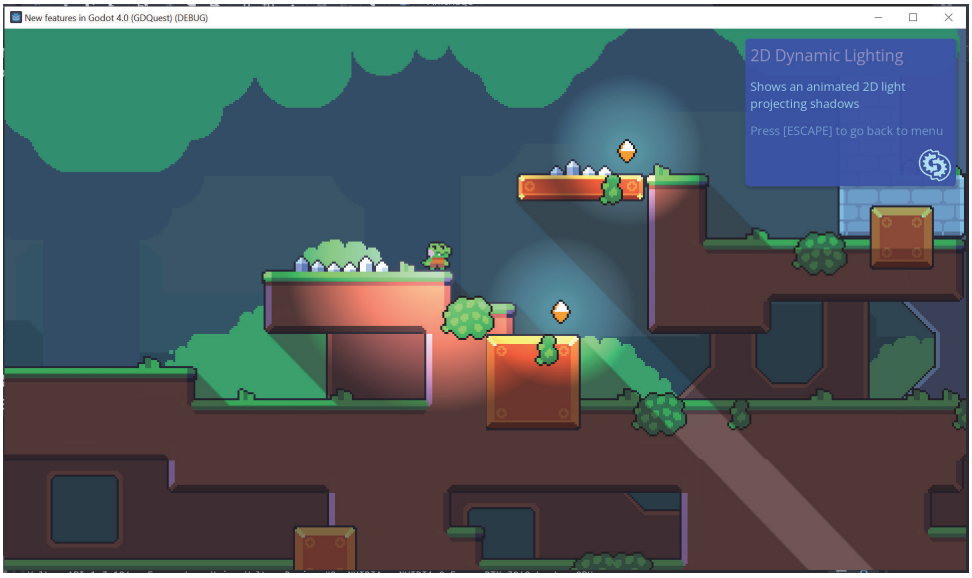
  Avec la sortie de la nouvelle version de Godot, des projets d'exemple ont été publiés sur GitHub. Ces projets sont accessibles gratuitement et l'intégralité du code source est ouvert. Vous pouvez ainsi étudier comment ils ont été faits. Vous trouverez dans le dépôt de nombreuses démos, 2D ou 3D. Ces démos exploitent les nouveautés de Godot et permettent de les voir en action.

Figure 28.1 : *Quelques démos du projet*



Pour accéder aux démos, téléchargez simplement le contenu du repo GitHub. Ouvrez ensuite le projet sous Godot et lancez la scène que vous souhaitez. Si vous voulez jouer aux démos, vous verrez qu'il y a une scène `main/ui_scene_selector` qui vous permet d'accéder à un menu de sélection. Ici vous pourrez lancer la démo de votre choix.

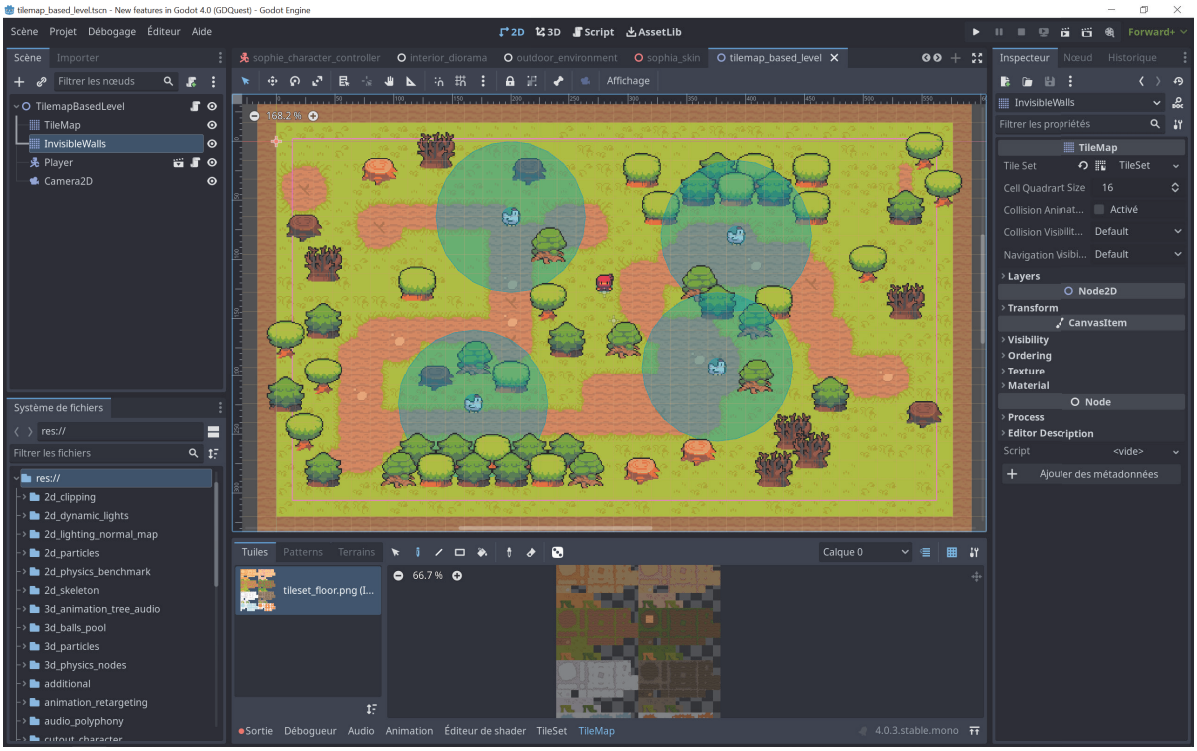
Figure 28.2 : *Lancement d'une démo*



Chaque démo correspond à un sous-dossier. Si vous ouvrez le sous-dossier correspondant à la démo en question, vous pourrez accéder à la scène, aux ressources

et au code de la démo. Vous pouvez alors éditer la scène et étudier comment elle a été conçue.

Figure 28.3 : *Analyse de la démo*



- Enfin, vous pouvez continuer à faire évoluer les deux projets que nous avons créés ensemble. Comme vous connaissez parfaitement ces projets, ce sera facile pour vous de vous y retrouver. Ajoutez-leur de nouvelles fonctionnalités, implémentez vos propres idées et créez différents niveaux afin d'aboutir à un projet complet. Cela vous demandera du temps mais une fois terminé, vous aurez acquis des automatismes et vous serez plus à l'aise avec Godot.

Avec ces quelques pistes, vous avez tous les éléments en main pour poursuivre votre apprentissage et vous lancer dans le développement de vos propres jeux vidéo avec Godot.

Concepts et termes anglais

Note *› Les traductions entre parenthèses sont mot à mot et fournies juste à titre indicatif.*

A

Assets

Ressources
Fichiers correspondant à des textures, sons, scènes, scripts, modèles 3D, toutes sortes d'éléments graphiques ou sonores nécessaires à la réalisation d'un jeu.

C

Collider

Zone de collision, Forme de collision
Désigne une forme (invisible pour le joueur) qui a pour but de rendre solide un objet de jeu. Dans la pratique, il définit la zone de contact qui déclenchera des interactions et simulera la solidité de l'objet. Voir Section 11.2, Solidité des tuiles et chapitres Interaction avec les objets ou Quelques objets à ramasser.

I

Inspector

Inspecteur
Désigne la zone de travail de Godot où vous pouvez visualiser et modifier les propriétés de l'objet actuellement sélectionné. Voir Section 1.2, L'interface de Godot, L'inspecteur.

Instantiate

Instancier
Action de créer une instance d'un objet préalablement configuré. Instancier un objet permet de le réutiliser à plusieurs endroits sans avoir besoin de le recréer. Par exemple, les pierres précieuses réparties dans un niveau sont des instances de la pierre créée par le développeur.

L

Label

Libellé, texte

Quand on parle d'interface utilisateur, un label correspond à un texte permettant d'afficher une information à l'écran comme le score, le pseudo, un dialogue...

Layer

Couche, calque

Regroupement virtuel d'éléments afin de pouvoir les manipuler par lot ou leur attribuer des caractéristiques communes. Le mot anglais *layer* signifie *couche* en français, mais en infographie le terme de *calque* lui est généralement préféré. Sur l'interface en français de Godot, selon le contexte, vous trouverez aussi bien couche, calque et layer.

M

Material

(Étoffe, matière)

En modélisation 2D ou 3D, composant qui définit l'apparence d'un objet, notamment sa couleur ou sa matière. Il permet de "colorer" les modèles avec des textures et des effets visuels. Le material permet également de décrire comment doit se comporter la lumière au contact de l'objet. Il n'y a pas vraiment de traduction française pour ce mot, mais traduire par matière se rapproche de la réalité. Le material correspond à la matière et donc à comment il nous apparaît visuellement.

Mesh

Maillage

En modélisation 3D, réseau de polygones qui définit la forme d'un objet. Dans Godot, les nœuds suffixés Mesh contiennent une propriété MESH qui permet de définir leur forme. Elle prend pour valeur un ensemble de formes élémentaires (ex: BoxMesh, SphereMesh).

N

Node

Nœud

Les nœuds sont les éléments fondamentaux pour la création de jeux. Ils ont un nom, des propriétés et peuvent être orga-

nisés afin d'avoir des enfants. Voir Section 1.4, Nœuds et arbres et l'encadré Différents types de nœuds [nodes].

P

Physics

(Physique, mécanique)
Propriétés d'un moteur de jeu chargées de simuler différents comportements et mécaniques physiques comme la gravité.

Post-processing

Post-traitement
Consiste à appliquer un traitement à l'image une fois le rendu 3D calculé afin d'améliorer le rendu visuel d'un jeu ou d'appliquer un effet artistique.

S

Shape

(Forme)
Détermine le type de forme d'un objet. Dans Godot, les nœuds suffixés Shape contiennent une propriété Shape qui permet de définir leur forme. Elle prend pour valeur différentes formes géométriques élémentaires (ex: RectangleShape2D, CircleShape2D, BoxShape3D) appelées aussi des primitives.

Signal

Signal
Correspond à un message envoyé par un nœud à un autre nœud lorsqu'un événement (clic, collision, condition) se produit. L'objectif est de déclencher un certain nombre d'événements à la réception de ce signal.

Sprite

(Visuel 2D)
Un sprite est une image permettant de définir l'apparence d'un objet ou d'un personnage 2D. Dans le cas d'un personnage, on utilisera plutôt une feuille de sprites, contenant différentes déclinaisons du visuel selon sa position ou son mouvement. La variante Sprite3D permet d'afficher une texture 2D dans un environnement 3D. En français, on utilise également le mot sprite. Il n'y a pas de réelle traduction.

T

Texture
: Texture
Image qui peut être déposée sur un objet 3D pour définir son apparence (on peut par exemple faire glisser une texture de brique sur un cube afin de créer un mur réaliste).

TileSet
: Collection/Planche de tuiles
Ensemble de tuiles carrées élémentaires permettant de créer des décors de jeux quand elles sont assemblées. Voir chapitre Mise en place d'un TileSet.

TileMap
: (Carte de tuiles)
Niveau créé avec des tuiles. Voir chapitre Mise en place d'un TileSet.

Trigger
: Déclencheur
Désigne une forme (invisible pour le joueur) qui a pour but de détecter une collision entre la forme et le joueur afin de déclencher un événement au moment de la collision. Contrairement au collider qui est une forme solide, le trigger est traversable.

U

UI
: Interface utilisateur
Désigne les composants (images, textes, formes) destinés à concevoir l'interface utilisateur comme un menu, une barre de vie, une barre de compétences, etc.

V

Viewport
: Fenêtre d'affichage
Désigne la zone de travail de Godot où vous assemblerez les éléments de vos scènes. La partie visible correspond à ce que la caméra active voit. Voir Section 1.2, L'interface de Godot, Fenêtre principale.

Index

Symboles

2D/3D, 8

A

Affichage, 223
Albedo, 202, 212
Aligner des éléments, 156
Ancrage, 130, 158, 264
AnimatedSprite2D, 83
Animation, 45, 82, 256
  gestionnaire, 13
Antialiasing, 225
Anticrénelage, 225
ApplyCentralImpulse, 241
Arbre, 16
Area, 218, 248
Area2D, 119, 139
  fonctions spécifiques, 122
area_entered, 249
Arrière-plan, 232
Assets, 9, 62
  (voir aussi Ressources)
Audio
  charger fichier, 153
  formats supportés, 149
  système de mixage, 13
AudioStreamPlayer, 149
AudioStreamPlayer2D, 152
AudioStreamPlayer3D, 272

B

Blender, 168
  coupe en boucle, 184
  créer un objet, 173
  dupliquer, 188
  exporter pour Godot, 198
  extrusion, 186
  mode Transparence, 194
  modes, 174
  redimensionner, 176
  rotation, 175, 190
  sélection, 179, 196
  sélection multiple, 186
  supprimer un objet, 173
  vues, 189, 195
body_entered, 122, 141, 143, 258
bool, 46
bounce, fonction, 144
Bouton, 157
Button, 49

C

C#, 4, 41
Calque
  de collision, 103, 120
  emplacements par défaut, 156
Caméra, 110, 222, 245
Canvas, 130
CanvasLayer, 130, 156
CharacterBody2D, 69, 76, 93, 142
Charger
  ressource, 153, 154
  scène, 160
  un script, 261
Chronomètre, 268, 268
Ciel, 232
class, mot clé, 44
Collada, 198
Collider, 105
Collision, 28, 79, 103, 119, 144, 217
  générateur, 205

CollisionObject2D, 25
CollisionPolygon2D, 92
CollisionShape, 205
CollisionShape2D, 27, 69
Commentaire, 47
Compilation, 162, 276
Compteur, 264
Condition, 47, 78, 78
Connecter un signal, 50, 250
Console, 48, 125
Constante, 46
Conteneur, 156, 158
Control, 24, 264
Coordonnées dans Godot, 68
Couleur, 202
.cs, extension, 73

D

.dae, 198
Débogueur, 13
Déclencheur, 247
delta, 45, 94, 240
Déplacer
    objet, 29
    PNJ, 144
Détection
    collision, 249
    fin du niveau, 248
    touche appuyée, 77, 239
Disposition, 158
Documentation, 80
Dupliquer, 257
DynamicFont, 159

E

Éclairage, 227
else, mot clé, 47, 78
Environnement de développement, 3
Événement
    area_entered, 249
    body_entered, 122, 141, 258
    pressed, 50, 160
    timeout, 269
Exécutable, 164, 278
[export], 239
Exporter, 164, 277
    pour Godot, 198
Extension
    .cs, 73
    .dae, 198

F

Fenêtre principale, 13
float, 46, 76
Fonction, 45
Format
    audio supporté, 149
    modèle 3D supporté, 198

G

GDScript, 41
Gestionnaire de projets, 5
GetAxis, 239
GetNode, 87, 125
GetParent, 125
GetVector, 77
glTF 2, 198
Gravité, 31, 93
    désactiver, 255
Grouper des nœuds, 30
GUI, 129
    (voir aussi Interface utilisateur)

I

Idle, 83
if, mot clé, 47, 78
Image
    au lancement du jeu, 163

de fond, 112, 156
Importer, 65
police d'écriture, 132
ressources, 10, 20, 64, 200, 214
sons, 152
Indentation, 46
input.GetVector(), *77*
Inputs, 41, *73*
Inspecteur, 12
Instanciation, 33, 90, *257*
int, 46
Interface Godot, *15*
Interface utilisateur, 129, 156, 264
Inventaire, 260
IsActionPressed(), *77*
IsOnFloor(), 93
itch.io, *278*

J

Jouer un son, 150

K

Krita, 212

L

Label, 49, 132
LabelSettings, 134
Lancer la scène en cours, 31, 39
Langue de l'interface, *5*
Lissage, *225*
Load, fonction, 153
Loop, 150
Lumière, *227*

M

Maillage, 205, 211
Masque, 120
Masque de collision, 103, 120
Material, 201
Menu principal, 156
Mesh, 205
MeshInstance, 211, 254
Mixage audio, 13
Modèle d'exportation, 165, *277*
Modélisation 3D, 183
Moteur de jeu, 2
Motion, propriété, 116
MoveAndCollide, 144
MoveAndSlide, *79*
MoveToward, *79*
Musique, 149
en boucle, 150

N

Navigation, 14
Niveau, 108, 208
déclencher fin, *247*
modéliser, 183
Node, 16
(voir aussi Nœud)
Node2D, 24
Normalized, *240*
Nouveau projet, *5*
Nouvelle scène, 18
Nœud, 16
créer, 23
différents types, 24
masquer, 31
principal, 36, 43
récupérer, 125, 266
renommer, 25
signal, 123
verrouiller enfants, 30, *71*

O

Objet
à ramasser, 254
couleur, 202

déplacer, 14, 29
détruire, 259
instancier, 257
paramètres, 22
physique, 23
propriétés, 11
redimensionner, 29
réutilisable, 19
taille, 22
visibilité, 26
Ombre, 227, 230
Origine (position), 68
override, 45

P

Panel, 42, 49
Parallaxe, 112
Paramètres
d'objet, 22
d'un objet, 11
du projet, 7, 162, 207, 225, 261, 276
Personnage, 69
déplacement, 144
instancier, 90
non joueur, 142
Physics Layer, 103
_PhysicsProcess, 76, 144, 240
PhysicsBody2D, 25
Physique, 23, 205
Play, fonction, 88
Playtest, 8
Police d'écriture, 132, 159
personnalisée, 265
PolygonShape, 92
Position, 22, 29
élément d'interface, 130, 158
Post-processing, 231
Post-traitement, 231
pressed(), 50, 160
Print, fonction, 48
_Process, 45, 76
Programmation, 41
Projet
créer, 5, 18
exporter, 276
paramètres, 7, 225
(voir aussi Paramètres de projet)
Propriétés d'un objet, 11
public, mot clé, 45

Q

QueueFree, 125, 259

R

_Ready, 45, 153, 245
Rebond, 31, 144
Recharger la scène, 142
Récupérer
axe de déplacement, 239
direction, 77
direction opposée, 144
nœud, 125, 266
touche clavier, 77, 239
Redimensionner, 29
Ressources, 9
(voir aussi Assets)
charger depuis un endroit donné, 153
importer, 10, 64
livre, 62
return, mot clé, 45
RigidBody, 23
RigidBody2D, 23, 24
RigidBody3D, 211
Root node, 23
Rotation, 22
animation, 256

S

Saut, 96
Scale, 22
Scène, 11, 18
  à partir d'un modèle 3D, 200
  charger, 160
  créer, 18
  de démarrage, 276
  instancier, 37
  lancer, 31, 39
  par défaut, 162
  recharger, 141, 252
  sauvegarder, 25, 71
Script, 41, 49
  ajouter, 43, 238
  charger automatiquement, 261
  global, 260
Shape, 28
Signal, 50, 123, 141, 143, 249, 258
Skybox, 232
Son, 149
  émettre un, 272
  importer, 152
  jouer un, 150, 154
  spatialisé, 152
  suite à un événement, 154
  volume, 150, 151
Spatial, 24
SpatialMaterial, 212
Sprite, 21, 26
  symétrie horizontale, 89
Sprite2D, 69
SpriteSheet, 82
StandardMaterial3D, 201
StaticBody, 34, 205
StaticBody2D, 34
string, 46
Système de coordonnées, 68
Système de fichiers, 9

T

Taille, 22, 29
Temps, 268
Texte, 49, 132, 157
  convertir nombre en, 136
TextEdit, 54
Texture
  ajouter à la scène, 21
  ancrage, 130
  créer, 212
  étendre, 157
  importer, 10, 20, 214
TextureRect, 156
Threads, 76
TileMap, 108
TileSet, 98
timeout(), 269
Timer, 268
Tonemaping, 234
ToString, fonction, 136, 267
Touche clavier, 77, 239
Transform, 21, 22
Triangle jaune, 26
Tuiles, 98

U

ui_accept, 96
ui_down, 74
ui_left, 74
ui_right, 74
ui_up, 74
using, mot clé, 44

V

Variable, 46
  modifier à la volée, 239
Vecteur, 68, 77
Verrouiller les nœuds enfants, 30, 71
Viewport, 13, 81

Visibilité, 26, 69
    nœud, 31
Visual scripting, 41
Vitesse, 240
void, 45
Volume du son, 151
Vues, 222

W

Workspace, 8
WYSIWYG, 5